# Un langage de programmation pour composer l'interaction musicale : la gestion du temps et des événements dans Antescofo

Jose-Manuel Echeveste

**THÈSE DE DOCTORAT DE**
**l'UNIVERSITÉ PIERRE ET MARIE CURIE**

Spécialité

**Informatique**

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

**José-Manuel ECHEVESTE**

Pour obtenir le grade de

**DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE**

Sujet de la thèse :

# Un langage de programmation pour composer l'interaction musicale

*La gestion du temps et des événements dans Antescofo*

soutenue le 22 mai 2015

devant le jury composé de :

| | |
|---|---|
| M. Jean-Louis Giavitto | Directeur de thèse |
| M. Arshia Cont | Encadrant de thèse |
| M. Pierre Cointe | Rapporteur |
| M. Miller Puckette | Rapporteur |
| Mme. Béatrice Bérard | Examinatrice |
| M. Gérard Berry | Examinateur |
| M. Dumitru Potop-Butucaru | Examinateur |
| M. Marco Stroppa | Examinateur |

José-Manuel Echeveste : *Un langage de programmation pour composer l'interaction musicale,* La gestion du temps et des événements dans Antescofo, © Présentée et soutenue publiquement le 22 mai 2015

# RÉSUMÉ

La musique mixte se caractérise par l'association de musiciens instrumentistes et de processus électroniques pendant une performance. Ce domaine soulève des problématiques spécifiques sur l'écriture de cette interaction et sur les mécanismes qui permettent d'exécuter des programmes dans un temps partagé avec les instrumentistes.

Ce travail présente le système temps réel *Antescofo* et son langage dédié. Celui-ci permet de décrire des scénarios temporels où des processus de musique électronique sont calculés et ordonnancés en interaction avec le jeu d'un musicien ou plus généralement avec un environnement musical. Pour ce faire, *Antescofo* couple un système de suivi de partition avec un système réactif temporisé.

L'originalité du système réside dans la sémantique temporelle du langage adaptée aux caractéristiques critiques de l'interaction musicale. Le temps et les événements peuvent s'exprimer de façon symbolique dans une échelle absolue (en secondes) ou dans des échelles relatives à des tempos.

Nous présenterons les domaines de recherche apparentés à *Antescofo* en musique, en informatique et en informatique musicale. Nous aborderons les caractéristiques du langage et de la partie réactive d'*Antescofo* qui ont été développés pendant cette thèse en particulier les stratégies synchronisations et les différents contrôles du temps et des événements permis par le système. Nous donnerons une sémantique du langage complet qui formalise le fonctionnement original du moteur d'exécution. À travers une série d'exemples d'applications issues de collaborations artistiques, nous illustrerons les interactions temporelles fines qu'il faut gérer entre une machine et un instrumentiste lors d'un concert. Le système *Antescofo* a pu être validé à travers de nombreux concerts par différents orchestres d'envergure internationale.

# PUBLICATIONS


[1] Arshia Cont, José Echeveste, Jean-Louis Giavitto, and Florent Jacquemard. Correct Automatic Accompaniment Despite Machine Listening or Human Errors in Antescofo. In *Proceedings of International Computer Music Conference (ICMC)*, Ljubljana, Slovenia, September 2012. IRZU - the Institute for Sonic Arts Research. URL http://hal.inria.fr/hal-00718854.

[2] José Echeveste. Stratégies de synchronisation et gestion des variables pour l'accompagnement musical automatique. Master's thesis, UPMC, Paris, 2011.

[3] José Echeveste, Arshia Cont, Jean-Louis Giavitto, and Florent Jacquemard. Formalisation des relations temporelles entre une partition et une performance musicale dans un contexte d'accompagnement automatique. In *Colloque Modélisation des Systèmes Réactifs (MSR)*, Lille, France, November 2011.

[4] José Echeveste, Jean-Louis Giavitto, Arshia Cont, et al. A dynamic timed-language for computer-human musical interaction. Technical report, Ircam-INRIA, 2013.

[5] José Echeveste, Florent Jacquemard, Arshia Cont, and Jean-Louis Giavitto. Operational Semantics of a Domain Specific Language for Real Time Musician-Computer Interaction. *Discrete Event Dynamic Systems*, 23 :343–383, December 2013.

[6] José Echeveste, Jean-Louis Giavitto, and Arshia Cont. A dynamic timed language for musician-computer interaction. *TOPLAS*, 2015 (submitted).

[7] Jean-Louis Giavitto, José Echeveste, et al. Real-time matching of antescofo temporal patterns. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, 2014.

[8] Jérôme Nika, José Echeveste, Marc Chemillier, and Jean-Louis Giavitto. Planning Human-Computer Improvisation. In *Proceedings of ICMC-SMC*, 2014.

[9] Christopher Trapani and José Echeveste. Real time tempo canons with antescofo. In *International Computer Music Conference*, page 207, 2014.

# TABLE DES MATIÈRES


1 INTRODUCTION 1
- 1.1 Le contexte musical 1
- 1.2 Partager le temps entre l'homme et la machine 2
- 1.3 Le projet Antescofo et son évolution 3
- 1.4 Contributions 6
- 1.5 Organisation de ce manuscrit 7
  - 1.5.1 Etat de l'art 7
  - 1.5.2 Le langage Antescofo 8
  - 1.5.3 Antescofo dans la pratique 9
  - 1.5.4 Perspectives et Annexes 10

I ÉTAT DE L'ART 11

2 LE TEMPS ET L'INTERACTION DANS LA MUSIQUE MIXTE 13
- 2.1 Les systèmes interactifs musicaux 13
- 2.2 La musique mixte 14
  - 2.2.1 Un bref historique 15
  - 2.2.2 L'école temps réel 15
  - 2.2.3 Critiques de l'école temps réel 16
  - 2.2.4 Partitions virtuelles 17
  - 2.2.5 Le problème de la notation dans la musique mixte 18
  - 2.2.6 Le problème de la synchronisation dans la musique mixte 18
- 2.3 Le suivi de partition 19
- 2.4 La modélisation du temps musical 20
  - 2.4.1 Du temps écrit au temps produit 20
  - 2.4.2 Les pivots temporels 21
  - 2.4.3 Les temps multiples dans la musique du XXème siècle 21
  - 2.4.4 Les structures temporelles dans la musique improvisée 22
- 2.5 Positionnement d'Antescofo 24

3 LES MODÈLES DU TEMPS EN INFORMATIQUE 27
- 3.1 Notion de modèle temporel 27
  - 3.1.1 Les systèmes continus 28
  - 3.1.2 Les systèmes événementiels 29
- 3.2 Succession 29
- 3.3 Simultanéité 31
  - 3.3.1 La simultanéité de durée zéro 31
  - 3.3.2 La simultanéité « bornée » 32
  - 3.3.3 L'absence de simultanéité 34
- 3.4 La datation des événements 35

3.5 Combiner plusieurs temps 36
3.6 Synchronisation 38
3.7 Positionnement d'Antescofo 40
4 LES USAGES DU TEMPS EN INFORMATIQUE MUSICALE 43
4.1 Le temps audio-numérique 43
4.2 Le temps réifié de la composition 45
4.3 Le temps performatif 46
4.4 Jouer une séquence dans le temps 50
4.4.1 Calcul audio en temps réel 50
4.4.2 Associer le temps symbolique au temps physique 51
4.5 Jouer des scénarios dans le temps 52
4.5.1 Organiser un scénario dans le temps 52
4.5.2 Le temps du live coding 54
4.6 Positionnement d'Antescofo 55
II LE LANGAGE ANTESCOFO 59
5 LANGAGE 61
5.1 Introduction 61
5.1.1 Couplage entre un modèle probabiliste et un modèle réactif 61
5.1.2 Programmer les interactions musicales 62
5.2 Présentation succincte du langage 62
5.2.1 Événements instrumentaux à reconnaître 63
5.2.2 Actions 63
5.2.3 Délais 64
5.2.4 Tempo et pulsation 64
5.2.5 Instants logiques 64
5.2.6 Coroutines 65
5.3 Expressions 65
5.3.1 Valeur 65
5.3.2 Variables utilisateurs 67
5.3.3 Accès aux dates d'affectations 68
5.3.4 Variables du système 68
5.4 Actions 68
5.4.1 Actions atomiques 69
5.4.2 Actions composées 69
5.4.3 Les processus 72
5.4.4 Conditionnelle 73
5.4.5 Forall 73
5.4.6 La structure de contrôle whenever 74
5.5 Démarrer et arrêter des actions 74
5.5.1 Les motifs temporels 75
5.6 Communications avec l'environnement extérieur 76
5.6.1 Communication avec Max et PureData 77
5.6.2 Communication *Open Sound Control* 77

6 LA GESTION DU TEMPS DANS ANTESCOFO 79
6.1 Interprétation musicale et musique mixte 79
6.2 Tempo 80
6.3 Positions courantes 81
6.4 Stratégies de synchronisation et gestion des erreurs 81
6.4.1 Stratégies de synchronisation avec tempo non-adaptatif 83
6.4.2 Stratégies anticipatives d'adaptation du tempo 86
6.4.3 Conservatif vs progressif 89
6.4.4 Stratégies de rattrapage d'erreurs 90
6.5 Créer un référentiel temporel 92
6.6 Tempo calculé explicitement 94
6.6.1 Récursivité temporelle 95
7 SÉMANTIQUE 97
7.1 Construction des domaines 100
7.2 Traces temporisées 104
7.3 Programme temporisé et coroutines 105
7.4 Programme *Antescofo* par un programme temporisé 111
7.4.1 Syntaxe abstraite des `group` 111
7.4.2 Syntaxe abstraite des processus 112
7.4.3 Syntaxe abstraite des autres actions 112
7.4.4 Syntaxe abstraite des programmes 113
7.5 Fonctions auxiliaires 114
7.5.1 Évaluation des expressions 114
7.5.2 Sélection de la prochaine coroutine à évaluer 114
7.5.3 Sélection de l'événement associé à une action `@tight` 115
7.5.4 Traduction d'une durée relative à un contexte temporel en une durée en seconde 115
7.5.5 Traduction d'une durée en seconde en une durée relative à un contexte temporel 116
7.5.6 Evaluation du délais de tête d'un programme 116
7.5.7 Suppression des actions portant un label donné 116
7.6 État, trace et sémantique d'un programme 118
7.7 Équations sémantiques de la fonction **Eval** 119
7.8 Tempo défini implicitement par une stratégie anticipative 127
7.8.1 Dynamic target 127
7.8.2 Static target 130
7.9 Remarques sur la sémantique 130
8 IMPLÉMENTATION 135
8.1 Contexte du développement logiciel 135
8.2 Organisation générale du code 136
8.3 Analyse syntaxique 137
8.4 Coroutines 138
8.5 Moteur d'exécution 140

8.5.1 Notification d'événement 140
8.5.2 Gestions des différents temps et ordonnancement 141
8.5.3 Évaluation des expressions et gestions des environnements 142

III ANTESCOFO DANS LA PRATIQUE 146

9 QUATRE ÉCRITURES DE *piano phases* 149
9.1 Contrôle dynamique des délais 149
9.2 Contrôle de dynamique du tempo + processus 151
9.3 Avec suivi de partition et contrôle continu d'un flux audio 152
9.4 Avec suivi d'une pulsation et réinjection 153

10 ÉTUDE PRATIQUE DES STRATÉGIES DE SYNCHRONISATION 155
10.1 Setup 155
10.1.1 Les partitions 156
10.1.2 Les objets contrôlés 156
10.2 Le problème de la latence 158
10.3 Estimation du tempo du musicien 159
10.4 Les stratégies de synchronisation 160
10.4.1 Tempo constant, accelerando, ritardando 160
10.4.2 leader/suiveur 163
10.5 Alterner estimation progressive et conservative 163
10.6 Nocturne 164
10.7 Les conclusions 164

11 ACCOMPAGNEMENT AUTOMATIQUE 167
11.1 Accompagnement d'un flux MIDI 167
11.2 Programmer la coordination d'un flux continu 168
11.3 Utilisation des synchronisations anticipatives 169

12 CANON DE TEMPOS EN TEMPS RÉEL 171
12.1 Conclusion 177

13 SUPERPOSITIONS DE COUCHES TEMPORELLES HÉTÉROGÈNES 179
13.1 Exemple de superposition de couches indépendantes 180
13.2 Organisation hiérarchique d'un processus électronique 181
13.3 tempo 184
13.4 Modes de jeu 185
13.5 Exemple d'utilisation des cibles statiques 185

14 GESTIONS DYNAMIQUES DE PROCESSUS 187
14.1 Combiner Antescofo, Max et SuperCollider 187
14.2 Organisation hiérarchique des réactions 187
14.3 Synthèse granulaire coordonnée au temps de la performance 190
14.4 Conclusion 191

15 SYNCHRONISATION DE PROCESSUS DE GÉNÉRATION AVEC IMPROTEK 193
15.1 Problème musical 193
15.2 Réalisation 193
16 SYNCHRONISATION DE PROCESSUS DE GÉNÉRATION AVEC IMPROTEK 197
16.1 Problème musical 197
16.2 Réalisation 197
17 UTILISATION D'*antescofo* DANS UN GROUPE DE MUSIQUE ÉLECTRONIQUE 201
17.1 Une partition interactive 201

IV CONCLUSION 205

CONCLUSION 207
17.2 Concepts utiles pour la musique mixte 207
17.3 Langage réactif 208
17.4 Augmenter l'expressivité de l'accompagnement 209
17.5 La gestion des temps multiples 209
17.6 Gestion de la simultanéité 209
17.7 Syntaxe et interface graphique 210
17.8 Généraliser la machine d'écoute 210
17.9 Validation 210
17.10 Performance 211
17.11 Intégration du calcul audio 212
17.12 La partition augmentée comme nouvel outil de création et d'analyse 212
17.13 Vers un langage de coordination 213
17.14 Vers une autonomisation du système 214

BIBLIOGRAPHIE 215

V ANNEXES 231

# 1 INTRODUCTION

Un humain et une machine peuvent-il jouer de la musique *ensemble* ? Le travail de cette thèse s'attaque à cette question dans le cadre de la musique mixte, en explorant une direction particulière : comment accorder le temps de la machine à celui de l'homme et vice versa. Notre objectif est de proposer de nouveaux outils informatiques pour l'écriture du temps musical et sa production dans l'interaction, afin d'explorer de nouvelles dimensions créatives entre composition et performance.

Les pièces de musique mixte intègrent des programmes s'exécutant en temps réel et en parallèle, associant par exemple des processus de traitement du signal (transformation, spatialisation sonore), de contrôle (lancement d'un processus, changement de paramètres) ou encore de synthèse (synthèse par lecture d'échantillons, synthèse par modèles de signaux, synthèse par modèle physique). Ces programmes doivent être exécutés à des instants particuliers, ni trop tôt, ni trop tard, dans le temps symbolique de la partition. Ils doivent suivre une évolution temporelle musicale qui peut dépendre de celle du musicien.

Quel paradigme de programmation, quel langage, et quels concepts permettent d'exprimer au mieux ces contraintes temporelles ? Quelles sont les informations nécessaires et comment le système doit-il réagir pour assurer la cohérence musicale de la partie électronique ? Quels modèles adopter pour améliorer la musicalité de l'interaction et faire en sorte que les musiciens puissent s'exprimer dans les meilleures conditions ?

## 1.1 LE CONTEXTE MUSICAL

La partition musicale est un support d'écriture et de pensée à travers lequel un compositeur organise des objets et des événements musicaux dans le temps. Ces objets datés dans un référentiel propre à la partition peuvent être composés dans des structures hiérarchiques, polyphoniques et séquentielles. Pour un informaticien, ces entités musicales se présentent comme instantanées (changements de notes ou de nuances, appoggiatures, etc.) ou se définissent dans la durée (glissandos, crescendos, phrases, structures, etc.). Traditionnellement, ces objets entretiennent entre eux des relations temporelles qui sont relatives les unes aux autres. L'unité utilisée est la pulsation, et le tempo permet d'associer le temps de la partition au temps physique. Ainsi,

une blanche dure deux fois plus longtemps qu'une noire qui dure une demi-seconde à un tempo à 120 *pulsations par minute*.

Le compositeur utilise la partition pour mettre en forme ses idées compositionnelles et les transmettre aux musiciens qui vont *interpréter* la pièce. Dans le contexte de cette thèse, interpréter signifie exécuter dans le temps physique les indications du compositeur en les déformant éventuellement (par exemple les variations temporelles peuvent amplifier l'expressivité de la pièce) et en les complétant (la partition n'est généralement pas une représentation exhaustive de la performance). Le compositeur laisse parfois délibérément une part d'indéterminisme dans la partition. Des problèmes de notation peuvent également introduire un manque de précision dans la spécification des idées musicales. L'interprète doit s'approprier cette liberté afin d'exprimer au mieux la musicalité de la pièce.

Lorsque plusieurs musiciens jouent ensemble, ils utilisent différents mécanismes pour coordonner leurs parties. Les gestes sont souvent utilisés pour communiquer des informations temporelles à certains instants pivots (début ou fin d'une phrase, mouvements qui accompagnent un rubato, etc.). De plus, pendant la performance, la représentation du tempo interne chez chaque musicien leur permet d'anticiper leur geste pour l'exécuter au *bon moment*. Cette représentation s'ajuste constamment aux événements de l'environnement, dont l'importance est variable pour la synchronisation, pour que les musiciens puissent partager une prédiction commune.

Nous ajouterons à cela le fait que pendant l'interprétation d'une pièce des relations temporelles complexes existent entre les musiciens (meneur, suiveur, etc.) Ces stratégies de coordination sont souvent non-symétriques et dynamiques.

C'est dans cette dualité entre les temps symboliques de la partition et les temps de la performance dans la musique mixte et électronique que s'insère le travail réalisé au cours de ce doctorat.

## 1.2 PARTAGER LE TEMPS ENTRE L'HOMME ET LA MACHINE

De nombreux compositeurs et musiciens contemporains insèrent dans leurs compositions et performances des processus électroniques pour générer, transformer, analyser ou encore spatialiser du son, mais aussi pour contrôler des lumières, de la vidéo, des robots etc.

Dans ce contexte, les questions précédentes prennent une dimension toute nouvelle.

Le travail de cette thèse s'est attaché à développer des outils pour l'écriture de processus, définissant un scénario interactif sous la forme d'une partition augmentée. Cette partition spécifie à la fois la musique qui sera interprétée par un musicien, et les actions électroniques qui devront être réalisées par la machine en interaction avec l'humain.

Le compositeur doit alors faire face à de nouveaux paradigmes : comment écrire le temps des actions électroniques et comment accorder ce temps à l'interprétation humaine lors de la performance ?

Les réponses apportées par les outils existants dans le domaine de l'informatique musicale ne sont pas entièrement satisfaisantes (cf. chapitre 4). En effet, si les systèmes développés sont nombreux et de natures différentes, leurs spécificités les destinent à certains types de tâches : les outils qui permettent de manipuler le temps symbolique sont essentiellement destinés à un travail en amont du concert, et les outils pour la performance offrent souvent un modèle temporel trop rigide.

## 1.3 LE PROJET ANTESCOFO ET SON ÉVOLUTION

Notre réponse a consisté à concevoir et développer le langage du système *Antescofo*. Ce système couple un suiveur de partition qui permet à une machine d'être à l'écoute des actions humaines et un langage dédié qui permet la spécification des actions électroniques.

*Antescofo* est né des travaux de thèse d'Arshia Cont dans le contexte de création musicale de l'*Ircam*. Depuis les articles séminaux de Roger Dannenberg et Barry Vercoe en 1984, les recherches sur le suivi de partition ont toujours constitué un domaine très actif. En 2007, Arshia Cont propose pour la première fois de coupler une machine d'écoute avec un langage permettant de décrire les séquences d'actions dans le temps de la partition. L'originalité de ce couplage est de permettre l'écriture du déroulement temporel de processus électroniques, en coordination avec des instrumentistes, sans pour autant figer le temps de ces processus. À l'image d'une partition classique, les relations temporelles des processus sont virtuelles et ne se réaliseront qu'au moment de la performance en fonction de l'interprétation humaine.

L'objectif est de permettre au compositeur d'exprimer des comportements électroniques dans un temps musical partagé avec celui d'un musicien. Le système est rapidement utilisé dans plusieurs productions de l'*Ircam* et encore aujourd'hui c'est dans un dialogue permanent avec les utilisateurs que le système continue d'évoluer.

Mon travail de thèse a consisté à étendre et à généraliser le langage et son modèle temporel. Le temps dans le langage peut à présent être manipulé, transformé et calculé de différentes manières, en utilisant à la fois une logique événementielle et chronométrique. Les structures temporelles peuvent se composer de manière hiérarchique, les processus électroniques se créer dynamiquement, et les durées s'exprimer par des expressions arbitraires, correspondant à des référentiels temporels calculés qui s'adaptent en temps réel à la performance humaine.

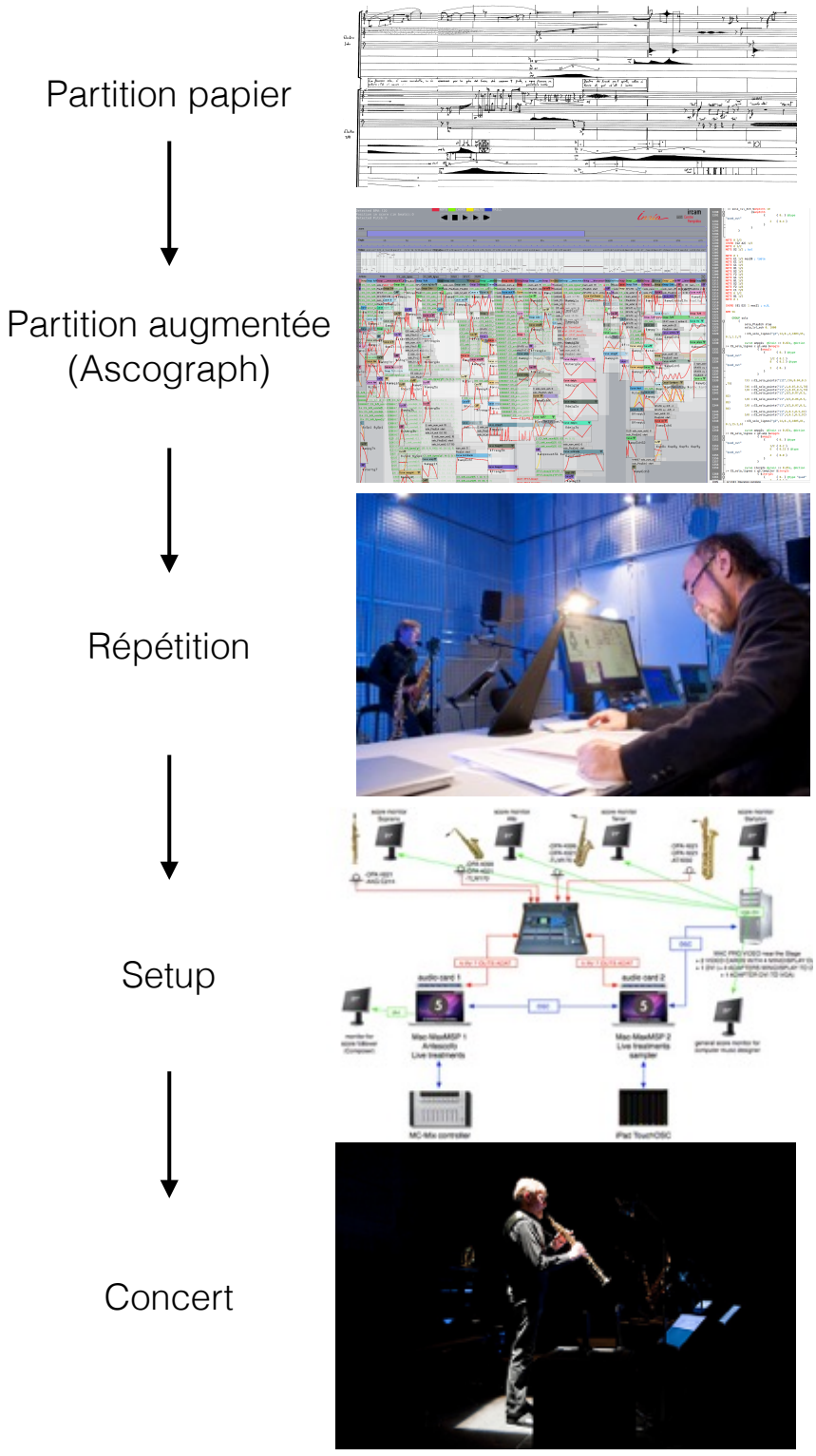



FIGURE 1: Schéma symbolisant les différentes phases de la création de la pièce de musique mixte *Iki-no-Michi* (2012) du compositeur Ichiro Nodaïra.

UN SCÉNARIO D'UTILISATION DU LOGICIEL ANTESCOFO. Un scénario d'utilisation typique correspond à la création d'une pièce de musique mixte avec suivi de partition pendant la performance (cf. Figure 1). En général, le compositeur écrit une première partition papier où il organise l'ensemble de la pièce (instruments + électronique). Il est parfois assisté par un *Réalisateur en Informatique Musicale* (RIM) pendant la phase de préparation et pendant le concert, pour la prise en main des différents outils informatiques et la réalisation de l'électronique. Des outils ont été développés dans *Ascograph*, l'éditeur associé à *Antescofo*, pour faciliter le transfert des partitions depuis les éditeurs de partition classiques vers le langage *Antescofo*. Le compositeur peut ensuite spécifier dans la partition *Antescofo* les processus temporels qui, pendant la performance, vont contrôler les différents paramètres de modules externes (synthèse, traitements, spatialisation, etc.) ainsi que la réception d'informations de l'environnement par le système. Ces modules sont pour la plupart codés dans un environnement de programmation tel que Max, CSound ou SuperCollider.

ÉVOLUTIONS DU SYSTÈME. Le développement du système et son évolution se font selon trois axes principaux :

- l'étude et l'application de techniques d'apprentissage automatique pour l'amélioration de la machine d'écoute ;
- l'étude et le développement de techniques de tests pour vérifier la cohérence des comportements temporels du système ;
- le développement du langage et du moteur d'exécution d'*Antescofo*.

C'est autour de ce dernier point que se situent les travaux réalisés au cours de cette thèse.

LE CONTEXTE « IRCAMIEN ». Cette thèse de doctorat s'est déroulée à l'*Ircam* au sein de l'équipe-projet *Inria Mutant*, elle-même rattachée à l'équipe *Représentations Musicales*. L'*Ircam*, Institut de Recherche et de Coordination Acoustique Musique, a été créé en 1969 par Pierre Boulez, avec l'idée fondatrice de réunir des artistes et des scientifiques autour de projets communs. Cet institut offre la possibilité aux équipes scientifiques de développer, d'expérimenter et de diffuser des outils en interaction directe avec tous les protagonistes du domaine :

- compositeurs,
- instrumentistes,
- réalisateurs en informatique musicale.

C'est dans ce cadre inédit de collaboration que s'inscrit mon travail de thèse.

Les productions musicales qui sont gérées par l'institut demandent un investissement considérable en ressources et en temps. Chaque création est prévue plus de deux ans à l'avance pour pouvoir mettre en place tous les éléments qui constituent un concert de musique

mixte. Une période de composition et d'investigation précède les nombreuses séances de tests avec et sans musicien, suivies des répétitions et des représentations publiques.

Pendant toute cette période, les équipes scientifiques concernées par le projet sont sollicitées pour aider à la prise en main des outils choisis, pour résoudre les problèmes qui apparaissent, et éventuellement pour développer de nouveaux mécanismes.

## 1.4 CONTRIBUTIONS

Mes travaux se sont concentrés sur l'étude et le développement du langage de programmation associé au logiciel *Antescofo*, permettant de décrire puis de réaliser un scénario dans lequel musiciens et ordinateur sont en interaction. L'objectif de ces travaux est de faciliter cette intégration dans un environnement compositionnel et performatif, où la gestion du temps joue un rôle central.

Ces problématiques m'ont amené à concevoir et à développer une notion de temps intégrant des aspects à la fois chronométriques et événementiels, permettant de définir des contextes temporels multiples, multi-échelles et hétérogènes, reliés par des relations élastiques. Ce modèle du temps se concrétise par des stratégies de synchronisation et de coordination et des mécanismes permettant de définir des tempos multiples au sein d'un langage de programmation temps-réel dédié.

Ma thèse a donnée lieu à quatre contributions théoriques principales :

- une étude des modèles de temps dans l'informatique et de l'informatique musicale (cf. chapitre 3 et 4) ;
- la définition de trois sémantiques du langage d'*Antescofo* (cf. chapitre 7) ;
- une étude des relations temporelles entre le musicien et la machine (cf. chapitre 6 et 10) ;
- une proposition d'un environnement pour la planification de scénarios dans le cadre des musiques improvisées (cf. chapitres 16 et 17).

Mon travail de recherche s'est aussi traduit par un investissement important dans la conception et les développements logiciels du système. À mon arrivée dans l'équipe en 2011, le langage *Antescofo* n'était qu'un langage de script, sans variable, sans expression, avec une seule stratégie de synchronisation pour les actions. Ma contribution a porté sur :

- l'analyseur syntaxique et lexical pour les partitions *Antescofo*,
- le moteur d'exécution pour la gestion des expressions et des actions,
- les mécanismes d'ordonnancement,
- les stratégies de synchronisation,

– les stratégies de rattrapage d'erreur,
– les interfaces de communication avec l'environnement extérieur.

Dans ce manuscrit nous nous focaliserons principalement sur mon travail concernant la gestion du temps dans *Antescofo* et sur les nombreuses études pratiques que j'ai menées afin de concevoir, expérimenter et valider les mécanismes temporels du langage.

Ce travail a en effet été validé en interaction constante avec des compositeurs, notamment une importante étude menée avec Marco Stroppa pour analyser les apports musicaux propres à chaque stratégie de synchronisation. J'ai aussi été amené à accompagner l'utilisation d'*Antescofo* dans le développement de plusieurs nouvelles pièces. J'ai ainsi pu :

– assister Julia Blondeau pour sa composition *Tesla ou l'effet d'étrangeté* ;
– concevoir et développer les mécanismes nécessaires à la gestion des relations temporelles nécessaires à l'expression de canons rythmiques et préparer les partitions *Antescofo* pendant la résidence de Christopher Trapani ;
– assister les compositeurs et RIM dans de nombreuses pièces parmi lesquelles *Iki-no-Michi* (Ichiro Nodaïra), *Re-Orso* (Marco Stroppa), *Partita II* (Philippe Manoury), *Dispersions de trajectoires* (José Miguel Fernández) ;
– concevoir et développer le dispositif scénique utilisé par le groupe *Odei* de musiques improvisées et électroniques, dispositif utilisé dans tous leurs concerts depuis 2013.

Chacune de ces œuvres fait appel de manière essentielle à des possibilités ouvertes par *Antescofo*. Elles ont permis de tester et de valider les mécanismes que nous avons introduits, dans le contexte d'un projet « en grandeur réelle ».

## 1.5 ORGANISATION DE CE MANUSCRIT

La question du temps sera le fil conducteur de ce manuscrit, du fait qu'elle tient une place centrale dans la conception, l'utilisation et l'exécution du sytème *Antescofo*.

### 1.5.1 *Etat de l'art*

C'est sous cet angle temporel que nous aborderons la première partie correspondant à l'état de l'art :

– quels sont les problématiques et les enjeux de l'écriture du temps en musique mixte ;
– sous quels points de vue le temps est pris en compte dans les langages de programmation ;
– quels sont les formes du temps qui apparaissent en informatique musicale.

Lorsqu'un compositeur utilise le langage *Antescofo*, c'est pour décrire une interaction qui prendra forme pendant le concert entre les instrumentistes et des processus électroniques. Au moment de l'écriture de ce scénario, le compositeur est capable de projeter dans le temps du concert différentes logiques temporelles (structure globale du morceau, séquence d'événements, tempos parallèles, temps élastique de l'interprétation, etc.). Les outils développés dans *Antescofo* lui permettent de manipuler ces concepts temporels dans le contexte de la musique mixte.

Nous présenterons dans le premier chapitre (chapitre 2) de l'état de l'art les problématiques du temps inhérentes à ce domaine en nous intéressant tout particulièrement aux questions de l'écriture de l'électronique et de sa synchronisation avec le musicien.

Pour synchroniser de la musique électronique avec le jeu des musiciens, il est nécessaire d'exécuter des programmes informatiques pendant la performance. Ces programmes s'exécutent à un instant donné et pendant une certaine durée. Le résultat du programme doit être prêt au bon moment musical, ni trop tôt, ni trop tard. Ils doivent donc respecter des contraintes temporelles qui peuvent être exprimées de différentes manières : « en même temps que », « après que », « pendant », etc. Cette problématique du temps a été largement étudiée en informatique. Le deuxième chapitre de l'état de l'art (chapitre 3) dresse un panorama des modèles de temps utilisés en informatique. Nous verrons l'importance de la conceptualisation du temps dans un langage sur la manière dont utilisateur réfléchit au problème à résoudre, et sur les propriétés des applications qui en découlent.

Les systèmes et les langages utilisés dans le processus de création musicale sont nombreux et de différentes natures. La plupart sont dédiés à un type de tâche (calcul audio, composition assisté par ordinateur, performance etc.) ou s'il ne le sont pas, leur conception induit au moins une gestion particulière du temps. Dans le dernier chapitre de l'état de l'art (chapitre 4) nous présenterons ces systèmes à travers la notion de temps qu'ils manipulent.

### 1.5.2 *Le langage Antescofo*

La musique mixte offre à l'informatique un champ d'étude riche et complexe, notamment par l'hétérogénéité des temps mis en jeu. Le système *Antescofo* s'appuie sur un langage dédié et un modèle temporel adapté aux contraintes du domaine, à travers une gestion unique des événements et de la durée.

La deuxième partie est dédiée à la présentation du langage *Antescofo*. Dans le chapitre 5 nous introduirons les concepts du langage qui permettent de décrire l'attente que l'on a d'un environnement musical incertain, ainsi que les structures de contrôles hiérarchiques existantes pour ordonnancer dans le temps des actions électroniques.

Les mécanismes temporels qui permettent la synchronisation de ces structures avec le jeu du musicien seront introduits dans le chapitre 6. Nous verrons en particulier comment on peut déduire des estimations de positions continues à partir des événements discrets reconnus par la machine d'écoute, ou créer son propre référentiel temporel. Nous étudierons les différentes manières de calculer une position locale à une structure pour l'exécution de ces actions dans le temps de la performance. L'ensemble des outils proposés permet au compositeur de décrire finement l'évolution de couches multi-temporelles qui s'adaptent au temps élastique du musicien.

Nous donnerons dans le chapitre 7 une sémantique de trace dans un style dénotationnel qui reflète le modèle multi-temporel du système, avant de donner dans le chapitre 8 un aperçu de son implémentation.

### 1.5.3 *Antescofo dans la pratique*

Les travaux de cette thèse ont été fortement influencés par les nombreuses collaborations musicales qui ont permis d'appliquer en pratique les outils développés. Ces collaborations ont aussi permis de valider les notions et les mécanismes que nous avons introduits (ils sont utiles et utilisés) ainsi que leur réalisation (les implémentation correspondantes sont suffisamment efficaces pour être utilisées en vraie grandeur dans des concerts).

Nous décrirons dans la troisième partie de ce manuscrit plusieurs applications faisant usage des fonctionnalités d'*Antescofo* dans différents contextes d'application :

- Le chapitre 9 propose plusieurs implémentations de l'œuvre *Piano Phase* de Steve Reich. L'objectif est de donner une vue d'ensemble du langage et d'illustrer ses possibilités et sa puissance sur un exemple concret.
- Le chapitre 10 détaille les séances de tests réalisées avec le compositeur Marco Stroppa et le pianiste Florent Boffard pour l'étude et l'évaluation des mécanismes de synchronisation du langage ;
- Les développements pour les applications d'accompagnement automatique seront présentés au chapitre 11. Ils ont été testés en collaboration avec *l'Orchestre de Paris*.
- Le chapitre 12 décrit le travail réalisé au cours d'une résidence en recherche artistique, sur la réalisation en temps réel de canons rythmiques. Au-delà de cet exemple musicalement important, cette application a initié l'étude des stratégies d'anticipation qui ont donné lieu aux mécanismes de synchronisation utilisés aujourd'hui dans des applications plus simples d'accompagnement automatique.
- Les chapitres 13 et 14 montrent l'utilisation du langage dans la composition de deux œuvres de musique mixte qui requierent

un contrôle *fin* pour la synthèse de nombreux processus sonores et qui repose sur des mécanismes *dynamiques*.
- Le développement d'un logiciel dans le cadre des musiques improvisées est présenté dans le chapitre 16. Cet exemple montre qu'*Antescofo* peut être utilisé dans un contexte de partition ouverte ou de scénario d'improvisation. Le système est ici utilisé comme un oeil permettant d'implémenter la coordination des différents modules du système.
- Enfin le chapitre 17 introduit l'utilisation d'*Antescofo* pour la création du dispositif interactif utilisé par le groupe de rock électronique dans leur concert.

### 1.5.4 *Perspectives et Annexes*

La dernière partie présente les perspectives ouvertes par mon travail. Le lecteur trouvera en annexe des informations supplémentaires :
- La traduction directe de la sémantique du chapitre 7 en caml ;
- Quelques données expérimentales issues du travail réalisé avec Marco Stroppa et Florent Boffard.

Certains travaux que j'ai réalisés et qui ont déjà été publiés ne sont pas abordés dans cette thèse :
- La définition d'un langage de motifs temporels et leur compilation dans *Antescofo* [GE+14] ;
- Une représentation du fragment statique de la partition augmentée d'*Antescofo* en automates temporisés [EJCG13].

Première partie

# ÉTAT DE L'ART

Dans cette partie, nous présenterons notre domaine de recherche, les travaux apparentés au système *Antescofo* et les concepts qui ont influencé le développement de cette thèse avec le *temps* comme fil conducteur. Le chapitre 2 présente les problématiques de la musique mixte et de l'interaction musicale. Notre angle d'approche se focalise tout particulièrement sur les relations entre partition et performance. Le chapitre 3 dresse un panorama des différentes approches du temps dans les langages de programmation. Enfin le chapitre 4 aborde le traitement du temps en informatique musicale face aux problèmes posés par la musique mixte.

# 2

# LE TEMPS ET L'INTERACTION DANS LA MUSIQUE MIXTE

Ce chapitre présente les éléments essentiels du temps dans la musique à prendre en compte pour le développement d'un système interactif musical. Après un bref descriptif de ces systèmes section 2.1, nous nous intéresserons plus particulièrement aux problèmes de la synchronisation et de la notation dans la musique mixte 2.2, nous présenterons le système de suivi de partition en section 2.3 comme une solution au problème de la coordination des processus électroniques au jeu d'un musicien et nous verrons comment le temps musical peut être modélisé (section 2.4). Enfin, dans la section 2.5 nous positionnerons le système *Antescofo* face à ces problématiques.

## 2.1 LES SYSTÈMES INTERACTIFS MUSICAUX

L'expression *systèmes interactifs musicaux* regroupe des systèmes utilisés dans des contextes très différents tels que la musique mixte [Tif94], les installations sonores [CS06], la conception de nouveaux instruments [JGAK07], la robotique [Mur14], les orchestres d'ordinateurs portables [Tru07], etc. On trouve dans la littérature plusieurs tentatives essayant de définir et classifier ces systèmes [Dru09]. En 1977, Joel Chadabe propose un modèle caractérisant les systèmes interactifs avec une boucle de rétro-action similaire à certains modèles en automatique [Cha77] (cf. figure 2). Joel Chadabe, qui a développé la notion de *composition interactive*, définit un système interactif comme un système qui influence l'interprète-compositeur contrôlant lui-même ce système.

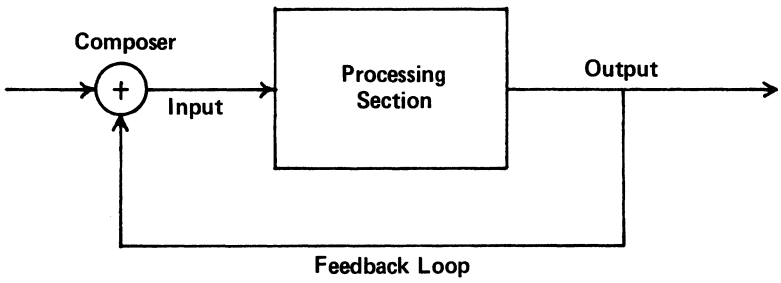



FIGURE 2: Modèle du sytème interactif extrait de l'article de Joel Chabade [Cha77].

Robert Rowe élargit la notion de compositeur-interprète et se focalise sur la réponse du système qui est à l'écoute et qui réagit à un environnement musical [Row92]. Il propose trois dimensions pour caractériser de tels systèmes :

- le système peut être orienté partition ou orienté performance ;
- la réponse du système peut être transformative, générative ou séquentielle ;
- le système repose sur un modèle de l'instrument ou un modèle de l'interprète.

Par ailleurs, il distingue dans les systèmes musicaux interactifs trois entités ayant trois fonctions caractéristiques différentes : la captation, le calcul et la réponse.

Todd Winkler [Win01] caractérise cinq tâches dans l'exécution d'un système interactif : le jeu du musicien, sa captation, l'interprétation des informations captées, l'exécution des programmes responsables de la génération et la sortie audio. Bert Bongers propose de classer les systèmes interactifs selon que l'interprète ou le public influence le comportement du système [Bon99]. Par rapport aux modèles de Robert Rowe et de Todd Winkler, Bert Bongers insiste comme Joel Chabade sur les boucles de rétro-action entre l'instrumentiste et le système (cf. Figure 3) mais aussi entre le public et le système.

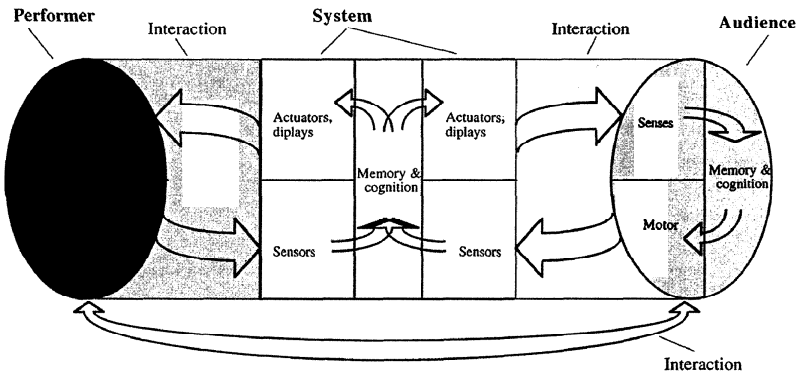



FIGURE 3: Modèle pour les systèmes interactifs extrait de l'article de Bert Bongers [Bon99].

## 2.2 LA MUSIQUE MIXTE

La musique mixte est un genre musical où l'interprétation d'une pièce donne lieu à l'association de musiciens humains avec des processus musicaux électroniques. Cette association fait intervenir une interaction forte entre humain et électronique et nécessite une coordination entre les deux, tant du point de vue compositionnel que performatif, qui a motivé des approches et des solutions variées.

Dans la suite de cette section nous allons esquisser le champs de la musique mixte du point de vue d'une de ses problématiques principales : la synchronisation de la musique électronique avec le jeu des musiciens. La musique mixte change les rapports traditionnels entre temps de la partition et temps de la performance. Ces questions seront abordées dans la section 2.4.

### 2.2.1 *Un bref historique*

*Imaginary Landscapes* n° 1 (1939) de John Cage peut être considérée comme la première œuvre musicale où un média d'enregistrement est utilisé dans la composition, jouant sur la confrontation entre le temps réel des instruments, et le temps différé du support enregistré. En 1951, Pierre Henry et Pierre Schaeffer du Groupe de Recherche de Musique Concrète (GRMC) monte pour la première fois une œuvre mixte. Il s'agit d'un opéra de musique concrète *Orphée51*. Bruno Maderna compose en 1951 *Musica su due dimensioni* pour flûte, percussions et bande, en collaboration avec Meyer-Eppler de l'Université de Bonn pour la réalisation de la partie électronique. Le compositeur met en avant un discours de « musique de chambre » entre un instrument et une bande magnétique, ouvrant la voie à l'étude d'un rapport dialectique entre les deux médias [Gri81].

Les schémas organisationnels typiques de l'œuvre mixte ont pour la plupart été énoncés par Karlheinz Stockhausen dans *Kontakte* (1959-1960) pour piano, percussions et bande, ainsi que dans *Mixtur* (1964) pour orchestre, générateur d'ondes sinusoïdales et modulateur en anneau. Pour la première fois, des transformations sonores de l'électronique sont réalisées pendant la performance en coordination avec l'orchestre.

Il faut comprendre que depuis les années 50, et dans une moindre mesure aujourd'hui, la grande majorité des pièces mixtes sont conçues pour un ou plusieurs instruments et une bande magnétique. Avec l'apparition d'ordinateurs suffisamment puissants, les bandes ont été remplacées par des fichiers numériques, mais elles restent d'une certaine manière fixes, obligeant l'interprète à se coller au temps du support. Cela n'a cependant pas empêcher la création de pièces exceptionnelles.

### 2.2.2 *L'école temps réel*

Le développement dans les années 80 des langages de programmation pour la musique ainsi que l'amélioration des techniques de synthèse et de traitement du signal vont ouvrir de nouvelles voies à la création musicale. Différents courants de pensée vont alors se

confronter [1]. L'école *temps réel* portée par l'Ircam avec des compositeurs tels que Pierre Boulez et Philippe Manoury vont promouvoir l'utilisation des traitements sonores en temps réel pendant la performance totalement intégré dans le processus compositionnel. Les possibilités offertes par les stations de calcul en temps réel à l'Ircam ont donné naissance à plusieurs œuvres dans des styles très différents, employant le temps réel. Philippe Manoury est sans doute le premier compositeur à avoir exploré de manière systématique les possibilités du temps réel. On lui doit le développement d'une véritable pensée du temps réel où une notion renouvelée de la partition joue un rôle central.

### 2.2.3 *Critiques de l'école temps réel*

Il est intéressant de noter que l'école *temps réel* a été l'objet de critiques constructives et intéressantes de la part notamment de Jean-Claude Risset [Ris99] et Marco Stroppa [Str99].

Risset attire l'attention sur le fait que les outils développés pour le temps réel concernent essentiellement la performance et souligne un manque de considération compositionnelle dans les environnements existants à l'époque. Il constate également la tendance des utilisateurs à employer le même genre de procédés techniques, ce qui engendre l'apparition de clichés.

Dans sa critique, Stroppa regrette que dans la musique qui se réclame de l'école *temps réel*, les efforts soient plus consacrés à des considérations technologiques qu'à des considérations esthétiques. Il s'intéresse au problème de la juxtaposition temporelle entre le jeu instrumental et électronique au cours de l'interprétation, et remet en question la valeur musicale ajoutée de l'électronique en temps réel en comparaison avec des pièces mixtes existantes, comme *Kontakte* de Stockhausen. Il met également en évidence la pauvreté des expressions musicales fournies par les outils de temps réel loin de la finesse du jeu d'un instrumentiste, surtout du point de vue temporel.

Le problème musical soulevé ici est à mettre en relation avec le fossé existant entre les logiciels pour la composition et ceux pour la performance mis en évidence par Puckette [Puc04]. Traditionnellement, la composition est un travail sur les structures musicales symboliques, notes, accords, phrases, harmonies et autres structures propres à chaque compositeur qui aboutit à la partition finale. Les temps y sont multiples, liés à des structures mises en œuvre au sein même du discours musical ou dépendantes du jeu de l'interprète, et de la liberté que le compositeur a donné grâce à des variations rythmiques plus ou moins définies. Il est légitime de se demander

1. Nous évoquons ici quelques exemples significatifs sans approfondir d'autres approches de la musique électronique en temps réel, telles que celles apparues dans les studios de Fribourg ou du STEM ou encore dans le rock et le jazz.

si la musique mixte basée sur un support fixe (interprète et bande) limite uniquement le jeu de l'interprète ou si d'une certaine manière, la partie compositionnelle est, elle aussi, « entamée » par cette rigidité technique. Autrement dit, dans quelle mesure l'expressivité du « langage » liée à ces bandes limite-t-elle le compositeur dans ses constructions temporelles ?

Dans la composition avec bande, l'écriture du temps peut être fine (structures temporelles complexes, constructions diverses dans l'écriture musicale de la partie électronique) mais la réalisation des sons électroniques peut se résumer à une question de montage. Il y avait alors une certaine dichotomie entre l'expressivité d'un langage qui sur le papier offrait beaucoup de liberté, et la rigidité temporelle de la bande une fois montée.

### 2.2.4 *Partitions virtuelles*

Au Moyen Âge, et dans une moindre mesure à la Renaissance, les partitions musicales n'étaient qu'un simple aperçu de la musique à jouer, le reste devant être déduit ou improvisé par l'interprète suivant des conventions souvent transmises oralement. Ce n'est qu'à partir de l'ère baroque que les compositeurs ont commencé à être plus précis en termes de hauteurs, de rythmes et d'articulations. Même si la précision des indications données à l'interprète a eu tendance à s'accentuer au cours du temps avec l'apparition d'indications de nuance, de timbre, de technique jeu instrumental, etc., certains objets musicaux tels que les dates de réalisation, les durées, les stratégies de synchronisation ou les nuances ne sont pas totalement déterminés ; cette indétermination laisse le champ libre à l'interprétation. C'est dans ce sens que Manoury développe le concept de partition virtuelle [Man90].

Une partition virtuelle est une organisation musicale dans laquelle on connaît la nature des paramètres que l'on souhaite traiter, mais pas leurs valeurs exactes, ces dernières étant exprimées en fonction de la performance du musicien. Selon Philippe Manoury, toute partition destinée à un interprète est dite « virtuelle » : elle ouvre des champs de possibles, sans les déterminer complètement.

Philippe Manoury étend ce concept à la partition virtuelle pour l'ordinateur : des programmes électroniques déterminés en temps réel en fonction d'un environnement musical. Autrement dit, un processus électronique existe dans la partition musicale, à côté de la transcription instrumentale et son résultat est évalué au cours de la représentation en fonction de l'interprétation instrumentale. Un objectif de la partition virtuelle est d'intégrer à la fois les aspects liés à la performance et ceux liés à la composition assistée par ordinateur dans un même cadre d'écriture.

### 2.2.5 *Le problème de la notation dans la musique mixte*

De nos jours, l'électronique d'une pièce mixte est composée de programmes s'exécutant en temps réel et en parallèle. Tous les paramètres de l'électronique, les paramètres d'écoute, d'interprétation, de timbres, etc. doivent être spécifiés par le compositeur s'ils ne sont pas intégrés dans les outils utilisés. La multiplication des paramètres de l'électronique implique à la fois une notation précise à l'intérieur des outils qui les emploient et le choix d'une représentation synthétique à même de résumer l'essentiel de l'identité musicale.

Il faut donc se poser la question des manières d'organiser et de noter un matériau qui demande autant de spécifications. La question de l'organisation du matériau rencontre ici celle de la notation. Celle-ci relève alors du choix compositionnel (telle prise de décision musicale étant susceptible d'impliquer telle ou telle notation) et non plus seulement d'un support représentatif neutre. Elle est aussi un système d'interprétation destiné à l'instrumentiste. Dans le cas particulier de la musique mixte ou de la musique électroacoustique, la question de l'interprétation d'une partition implique alors de poser la question de sa représentation : que représente-t-on et pour quel « interprète » (humain, non humain) ? Les nécessités de représentation diffèrent donc, selon que l'on se place du point de vue de la composition, de la réalisation ou de l'interprétation.

La partition de composition est le lieu de la pensée et de l'élaboration musicale dans lequel le degré d'expressivité de la notation est le plus déterminant. La notation doit ici représenter un réel tremplin pour l'imaginaire.

La partition de réalisation est quant à elle la plus précise, contenant toutes les informations nécessaires à l'élaboration finale de l'œuvre musicale. Cette partition est la plus exposée au problème de la prolifération des paramètres. La partition peut rapidement devenir illisible selon le degré d'hétérogénéité des éléments électroniques utilisés. La partition électronique finale peut avoir cela d'étrange qu'elle ne représente pas forcément le résultat sonore voulu mais plutôt la manière de le produire. On trouve cependant ce type de démarche dans le monde purement instrumental, chez Lachenmann notamment.

La partition d'interprétation, destinée à l'interprète, indique les éléments les plus essentiels et permet de ce fait un vrai travail de musique de chambre, lui indiquant par exemple les points de synchronisation et les zones plus floues avec lesquelles il peut jouer.

### 2.2.6 *Le problème de la synchronisation dans la musique mixte*

Les difficultés que rencontrent les compositeurs pour coordonner des sons électroniques avec le jeu d'un instrumentiste les obligent à considérer ces problématiques dès la conception de l'oeuvre. Souvent

un métronome est placé aux oreilles du musicien pour permettre la synchronisation avec l'électronique, comme dans *Lyric Variations for Violin and Computer* de Randall. Dans *Kontakte*, Stockhausen procède par succession de moments musicaux lui permettant de synchroniser l'électronique aux débuts et aux fins de ces moments. Le compositeur peut également choisir de fragmenter l'électronique, un opérateur étant responsable du lancement de chaque partie comme dans *Musica su due dimensioni* de Bruno Maderna, ainsi que dans le cycle *Synchronisms* de Mario Davidovsky. Ce séquencement est parfois réalisé par l'interprète lui-même par le biais d'une pédale ou d'un autre contrôleur. Même si des processus souvent plus complexes ont aujourd'hui remplacé la bande magnétique, cette dernière solution de synchronisation est d'ailleurs toujours d'actualité dans certaines pièces contemporaines.

## 2.3 LE SUIVI DE PARTITION

On définit traditionnellement le suivi de partition comme l'alignement automatique d'un flux audio correspondant à un ou plusieurs musiciens sur une partition symbolique. En suivant la partie du musicien le système doit être capable d'exécuter les commandes de l'électronique en s'adaptant aux variations de l'interprétation. La musique électronique peut désormais être interprétée.

Dannenberg et Vercoe ont été dans les années 1980 les premiers à proposer un système de suivi de partition [Dan84a, Ver84]. Les entrées sont symboliques (protocole MIDI) avant d'être étendues par la suite à des entrées audio.

Philippe Manoury a été l'un des premiers compositeurs à intégrer ces techniques dans ses oeuvres à la fois comme un processus de composition et comme un outil de performance. Il collabore notamment avec Miller Puckette pour les compositions de *Jupiter* (1987) et *Pluton* (1988 – 1989). Ils développent ensemble de nombreux principes d'interactivité en temps réel, prémisses de l'environnement de programmation Max [Puc91].

A la fin des années 1990, l'intégration des méthodes probabilistes, fondées sur des modèles de Markov cachés a amélioré la robustesse de ces systèmes. Nous ne présenterons pas plus avant les nombreux systèmes de suivi de partition qui ont été développés jusqu'à aujourd'hui. Un des derniers exemples est le système *Music-Plus-One* [Rap11], un suiveur adapté au répertoire classique et capable de synchroniser un fichier audio d'accompagnement au jeu du musicien d'une manière musicalement expressive. Ce système nécessite une phase d'apprentissage pour chaque interprète et chaque morceau et il n'est robuste que sur des instruments monophoniques. Comme les autres systèmes développés dans ce domaine, il n'est pas possible d'éditer ses propres partitions ni de programmer ses propres actions d'accom-

pagnement. Il n'y a donc pas de réflexion particulière autour du langage musical, ces sytème n'étant pas destinés à la composition de musiques interactives.

Plus récemment, de nombreuses applications dans le domaine du suivi de partition pour tablettes tactiles ont vu le jour comme Cadenza, Tonara (pour tourner automatiquement les pages de la partition) et MusicScore [LFL12] (pour l'édition de partitions et l'accompagnement automatique). Ces applications ciblent principalement des musiciens dans un but pédagogique, sont peu robustes en situation de concert et n'ont pas d'application dans la création musicale.

Les systèmes présentés sont spécialisés dans la tâche d'accompagnement automatique simple (coordination d'un fichier audio ou MIDI avec le jeu du musicien) et aucun ne propose un langage générique d'action.

## 2.4 LA MODÉLISATION DU TEMPS MUSICAL

Le temps en musique est bien évidemment une problématique qui excède très largement le cadre de cette thèse. Nous nous contenterons d'aborder le *temps musical* de manière très partielle et partiale sous l'angle de l'interprétation à travers la notion de temps multiple et de la coordination de ces temps.

### 2.4.1 *Du temps écrit au temps produit*

Les relations entre le temps qui est écrit dans une partition et le temps qui est produit lors d'une performance ont été largement étudiées au cours de ces dernières années.

Dans le domaine des sciences cognitives un très grand nombre d'études se sont attachées à décrire ou imiter les paramètres (les paramètres temporels, ceux liés aux dynamiques, aux articulations, etc.) qui définissent une interprétation musicale, en particulier à travers les représentations temporelles cognitives sous-jacentes. Plusieurs synthèses de la littérature ont été réalisées dans [WG04, KM09].

La relation entre partition et performance est souvent réduite à la problématique du tempo, même si le tempo reste une notion très discutée dans la communauté. Les variations de tempo peuvent reposer sur un système de règles [Fri95], des modèles cinématiques (fondés sur une analogie mécanique) [Tod95], des modèles mathématiques (sans justifications autres que formelles) [MZ94], des réseaux de neurones artificiels [Bre00] ou encore des modèles dynamiques d'oscillations neuronales [LJ11].

Les études perceptives montrent l'influence structurelle des pièces (hiérarchie, complexité, relation main gauche main droite pour le piano) dans le placement temporel du musicien [Pal97]. Elles soulignent l'importance de l'attente musicale comme un processus impli-

cite de prédiction et d'anticipation chez le musicien [LJ99]. D'autres montrent les différences dans les stratégies de synchronisation adoptées par les musiciens lorsqu'ils jouent à plusieurs en fonction des relations établies meneur/accompagnant [WEBV14] ou en fonction du retour auditif et visuel [ZPP15]. Plusieurs synthèses de recherches concernant les études perceptives et cognitives de la synchronisation sensorimotrice sont disponibles [Rep06, RS13].

Le problème de l'inférence d'un tempo (continu) à partir de l'observation d'événements musicaux discrets fait l'objet de nombreuses propositions reposant sur des outils formels ou mathématiques très divers allant du couplage d'oscillateur à un problème d'optimisation géométrique [Nou08].

### 2.4.2 *Les pivots temporels*

Marco Stroppa a développé une représentation originale du temps qui repose sur la synchronisation par pivots temporels [DS90] utilisée dans certaines œuvres telles que le troisième mouvement de *Traiettoria* (1984-1985). Le concept s'applique aussi bien à la musique mixte que purement instrumentale.

Il existe souvent dans les structures musicales temporelles des évènements qui présentent des points saillants, perceptivement plus pertinents que d'autres ; par exemple, l'attaque d'une note percussive ou la dernière note d'une phrase musicale. Lorsque le compositeur associe plusieurs structures temporelles, il utilise souvent ce genre de repères pour les coordonner. Ce mécanisme de coordination mis en évidence par Stroppa est un processus compositionnel facilement transposable au temps de la performance pour la réalisation de stratégies de synchronisation entre les musiciens ou entre les musiciens et l'électronique.

### 2.4.3 *Les temps multiples dans la musique du XXème siècle*

On retrouve dans de nombreuses œuvres du répertoire du XXème siècle la volonté chez les compositeurs de manipuler des temps multiples. C'est en particulier à travers des spécifications complexes de tempo que cette notion s'incarne.

*Pléïades* (1979) de Iannis Xenakis, *Tempus ex Machina* (1979) de Gérard Grisey sont des exemples de pièces où les musiciens évoluent avec des tempos différents. Dans *Kammerkonzert* (1970) de Ligeti, le chef d'orchestre indique différents tempos pour chaque musiciens.

Dans son quatuor n° 3, Elliott Carter divise l'ensemble en deux parties et attribue un tempo à chacune des deux parties. Pour réaliser ce

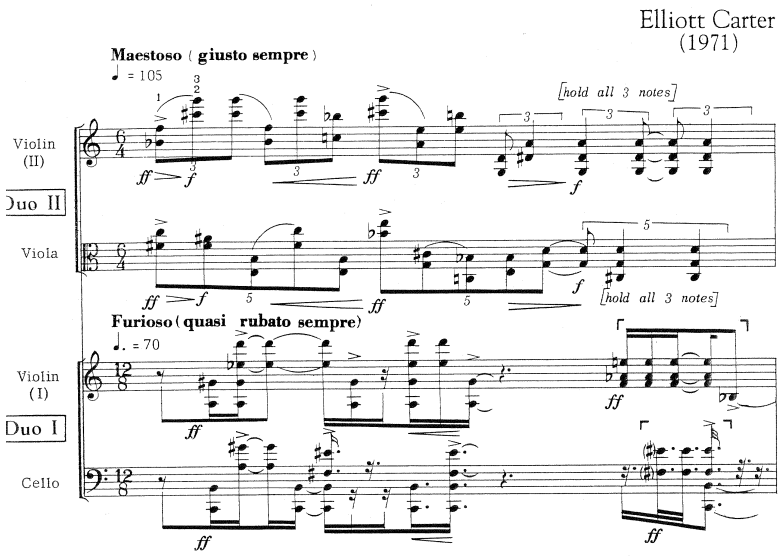



FIGURE 4: Extrait de la partition du Quatuor n° 3 d'Elliot Carter (1971).

contrepoint multi-temporel, le compositeur utilise du papier millimétré afin de calculer « à la main » les synchronisations (cf. Figure 4) [2].

La pièce *Gruppen* est sans doute l'exemple le plus spectaculaire où le compositeur, Stockhausen, développe cette idée de tempos multiples. Ici, trois orchestres dirigés par trois chefs d'orchestre évoluent à des tempos différents. C'est aux chefs d'orchestre d'assurer la synchronisation en des points précis entre les différents orchestres (cf. Figure 5).

Ces quelques exemples rapidement esquissés montrent l'importance de la problématique des temps multiples dans la musique contemporaine. Cette recherche musicale nécessite le développement d'outils adaptés pour faciliter l'expression de la pensée musicale en terme de tempo et de synchronisation dans la composition mais aussi dans la performance.

Dans cette direction, le chercheur-compositeur John McCallum a par exemple développé un outil d'aide à la composition pour calculer les courbes de tempos de voix polyphoniques qui respectent des contraintes de synchronisation de tempo et/ou de position [MS10].

### 2.4.4 *Les structures temporelles dans la musique improvisée*

Dans les sections précédentes, nous avons abordé le temps musical *écrit*. Cependant les structures temporelles ne se limitent pas à la musique écrite mais sont tout aussi essentielles dans la musique improvi-

2. Il s'agit ici de temps multiples coordonnés par modulation métrique (105 = 70/2*3), donc réductibles à un seul tempo de base correspondant à la technique de la modulation métrique chez Carter.

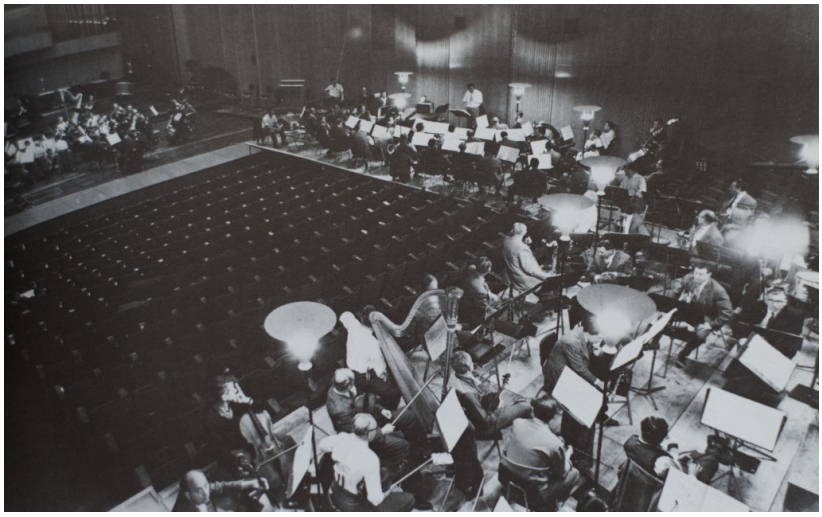


FIGURE 5: Répétition avant la première de Gruppen de Karlheinz Stockhausen, pièce pour trois orchestres (photographie : Heinz Karnine)

sée. En effet, la plupart des styles d'improvisation s'appuient sur une connaissance a priori de l'organisation temporelle de la musique à produire. Cette organisation temporelle peut être une séquence explicite comme dans les thèmes de standard de jazz ou des mélodies dans les musiques folkloriques traditionnelles. D'autres fois, la structure temporelle peut être spécifiée comme un scénario séquentiel décrivant une séquence de contraintes qui doivent être remplies par le ou les musiciens improvisateurs. Un exemple classique est la progression harmonique utilisée dans la plupart des styles musicaux occidentaux actuels tels que le rock, le blues, le jazz ou la musique pop.

Parfois, l'organisation d'une performance ne prend pas la forme d'une structure séquentielle mais est organisée à travers des réponses à des événements complexes impliquant à la fois des événements musicaux et des conditions logiques, comme par exemple dans le *soundpainting* [Dub06]. Ces différentes formes de structures temporelles peuvent se succéder ou même exister simultanément. La coordination de ces séquences et de ces réactions temporelles constitue des plans d'improvisation complexes ou des partitions dynamiques.

Il existe différents systèmes informatiques pour l'improvisation. Certains systèmes de « co-improvisation » créent de nouvelles improvisations en naviguant dans une représentation extraite du jeu d'un musicien capté en temps réel  comme dans les logiciel de la famille de OMax [ABC+06, LBA12]. D'autres systèmes tels que *Continuator* vont chercher à apprendre le style d'un musicien pour ensuite générer du nouveau matériel dans le style appris [Pac03]. *Voyager*, développé par George Lewis, est un système basé sur des règles qui génèrent en temps réel des réponses complexes au jeu d'un musicien improvisateur [Lew99]. Les derniers travaux de recherche en informatique

autour de l'improvisation en musique introduisent l'idée de scénario pour guider la génération [PRMd13, NC15, SD13, DLSW13] Cependant peu d'études en informatique musicale se sont concentrées sur la spécification et la gestion en temps réel de scénarios complexes dans le cadre des musiques improvisées interactives.

## 2.5 POSITIONNEMENT D'ANTESCOFO

*Antescofo* ne sera présenté que dans la partie suivante mais nous le positionons dès à présent dans le paysage et les problématiques de la musique mixte.

UN SYSTÈME INTERACTIF. *Antescofo* est un système musical interactif programmable. Bien qu'*Antescofo* soit souvent vu comme un suiveur de partition, donc *orienté partition* selon les axes de Robert Rowe, avec une réponse séquentielle et un modèle interprète, le développement du langage a considérablement étendu ses caractéristiques. En effet, le contexte de création musicale dans lequel s'est développé *Antescofo* a montré la nécessité de développer des outils permettant de décrire des scénarios interactifs complexes sortant du simple cadre du suivi partition. Ainsi le langage d'*Antescofo* est dédié à la création de structures temporelles complexes et il permet aujourd'hui de concevoir l'organisation temporelle d'applications pour la musique mixte, pour la musique improvisée, ou encore pour des installations interactives.

ANTESCOFO ET LE TEMPS RÉEL MUSICAL. La problématique du temps réel musical et ses critiques ont fortement influencé l'élaboration du langage temporel d'*Antescofo*. Le langage s'inspire de la relation complexe entre la partition et l'interprète et tente de donner aux compositeurs un médium riche qui leur permette d'exprimer les relations temporelles, y compris dans l'électronique seule, qu'ils imaginent à l'intérieur de leurs partitions. Il y a alors la possibilité d'une vraie dialectique entre couches temporelles : entre l'instrumentiste et l'électronique, mais aussi entre toutes les couches temporelles qui peuvent exister au sein même de l'électronique. La réalisation de l'électronique n'est alors plus seulement un geste final d'assemblage mais un vrai instrument de composition et de conceptualisation.

LA PARTITION AUGMENTÉE. Le concept de partition virtuelle a largement inspiré le caractère dynamique du langage ainsi que les considérations interprétatives du modèle d'*Antescofo*.

Les questions de notations se retrouvent dans *Antescofo* et mettent en jeu des structures de contrôle et de données permettant l'expression de ces différents types de notation. Les choix de ces structures

et d'une syntaxe adaptée influencent considérablement l'expressivité du langage.

LE SUIVI DE PARTITION. La machine d'écoute d'*Antescofo* est polyphonique [Con10] et décode en temps réel le tempo du musicien. La connaissance du tempo permet d'anticiper, à l'instar d'un musicien humain qui construit une échelle temporelle en fonction des évènements des autres musiciens et d'une notion musicale du passage du temps. *Antescofo* est le seul système de suivi qui permette explicitement de commander des modules externes, de gérer les erreurs et de traiter le hiatus entre tempo estimé, tempo réel, tempo idéal et événements reconnus. Par ailleurs, *Antescofo* ne nécessite pas de phase d'apprentissage sur l'interprétation à écouter, le rendant ainsi plus robuste aux changements de l'environnement (instrument, salle, micro, bruit, etc.). Pour plus de détails sur la machine d'écoute, nous invitons le lecteur à consulter [Con10].

Dans son utilisation en musique mixte, le module de suivi de partition d'*Antescofo* est responsable d'une coordination entre l'ordinateur et le musicien. Fidèle à la partition, il permet des interactions musicales complexes semblables à celles pouvant exister entre des musiciens.

LE TEMPO. La détection du tempo à partir d'un flux audio est rendu possible grâce à un modèle explicite du temps inspiré de modèles cognitifs de synchronisation musicale dans le cerveau [LJ99]. Les variations du tempo décodées en temps réel se répercutent automatiquement sur toutes les durées qui dépendent du tempo du musicien et permettent d'anticiper son évolution temporelle.

Nos développements restent cependant orthogonaux au modèle de tempo utilisé et par exemple le langage permet de définir son propre modèle de tempo. Ce modèle peut être combiné avec des stratégies de synchronisation dynamiques pour affiner le comportement temporel des actions d'accompagnement comme nous le verrons dans la partie suivante.

PIVOTS TEMPOREL S La notion d'événement pivot a motivé le développement de stratégies de synchronisation particulières où le compositeur peut spécifier des événements cibles pour guider la coordination des processus électroniques avec le musicien (cf. Chapitre 6). Ainsi, en prenant en compte à la fois le tempo et les pivots temporels, le modèle temporel d'*Antescofo* est fondé sur un modèle hybride qui gère à la fois temps discret des événements et des structures rythmiques, la durée et le temps continu des processus musicaux.

TEMPS MULTIPLES. Dans *Antescofo*, des mécanismes permettent de gérer les variations de tempos locaux. Ces tempos peuvent être

associés à chacune des entités musicales s'exécutant en parallèle et leurs variations peuvent être décrites dans leur propre référentiel ou dans un référentiel indépendant.

*Antescofo* étend ces problématiques de tempos multiples dans le contexte de la performance avec un environnement incertain. Le suivi de partition étant de plus en plus fiable, l'apport de ce langage « temporel » peut augmenter de manière conséquente les possibilités d'écriture de contrepoints temporels complexes et dynamiques, ne délaissant jamais la liberté de l'interprète tout en ne faisant aucun compromis d'un point de vue compositionnel.

ANTESCOFO DANS UN CONTEXTE DE MUSIQUE IMPROVISÉE. *Antescofo* offre des structures de contrôle qui permettent de déclencher des processus musicaux en dehors du temps noté d'une partition écrite à l'avance. Il offre donc des mécanismes adaptés au contexte dynamique de la musique improvisée permettant de planifier une performance qui met en jeu divers procédés musicaux. Ses mécanismes originaux de synchronisation, de réactions et de communications facilitent alors le développement et la réalisation de telles applications [NECG14].

# 3

# LES MODÈLES DU TEMPS EN INFORMATIQUE

Dans cette partie nous nous intéressons aux grandes familles de modèle de calcul à travers un point de vue particulier qui est la gestion du temps. Lorsque les actions de communications et calculs d'un système informatique sont coordonnées dans le temps, alors il existe un modèle temporel sous-jacent. Notre objectif est de faire un panorama des différentes approches développées dans les langages de programmation pour gérer les temps et situer *Antescofo* dans ce paysage.

Nous présentons les modèles temporels qui sous-tendent les langages de programmation sous la forme d'une classification qui reprend les deux « objets » classiques du temps : l'instant discret et la durée continue (cf. Figure 6). Le premier correspond au système événementiel et le second est associé à des systèmes continus. Nous aborderons les modèles qui mélangent les deux notions dans un troisième temps.

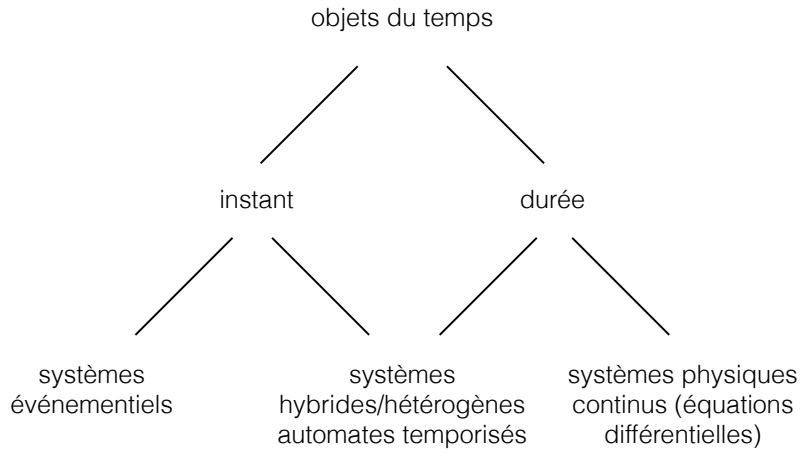



FIGURE 6: Notre classification des modèles de temps en Informatique

## 3.1 NOTION DE MODÈLE TEMPOREL

Tout calcul se déroule dans le temps mais les relations temporelles sont plus ou moins explicites ou interviennent plus ou moins dans le résultat du calcul. Un modèle de temps est une abstraction qui permet au compilateur d'expliciter les objets du temps (instants, dates, durées, événements, etc.) et les relations temporelles qu'ils entretiennent.

Plusieurs langages différents peuvent partager le même modèle temporel sous jacent. Par exemple C et Java ont le même modèle temporel. Inversement, suivant le niveau de détail ou la perspective adoptée, on pourra distinguer plusieurs modèles de temps à l'œuvre dans un langage de programmation. Par exemple, le modèle de temps qui permet de choisir la séquence d'instructions optimale du compilateur du langage C n'est pas le modèle de temps pertinent et utile pour le programmeur qui développe son application algorithmique [1].

Un bon modèle permet de se poser les questions simplement sur les problèmes à résoudre. Dans cette thèse, nous nous attachons à développer un modèle temporel nouveau dédié à l'expression des processus musicaux mixtes. Choisir le bon modèle pour une situation particulière est donc un problème délicat non seulement parce que le compromis d'abstraction devra être le bon mais aussi parce que le modèle repose et communique avec d'autres modèles de natures différentes.

### 3.1.1 *Les systèmes continus*

Les systèmes continus correspondent par exemple à des systèmes décrits par des ensembles d'équations différentielles qui modélisent les comportement physiques étudiés en automatique.

La musique nécessite de décrire des objets continus comme la durée d'une note, les variations d'amplitude ou de fréquence, etc. Les systèmes que nous considérons sont en interaction avec un environnement physique continu. Cependant les objets continus pertinents dans notre problématique se réduisent principalement à des durées et à la notion de tempo (cf. section 6.2) qui correspond à un analogue de la notion de vitesse.

De ce point de vue, si la position du musicien dans la partition à une date donnée peut se voir comme l'intégrale du tempo et donc faire référence à un formalisme différentiel, son usage sera très restreint et se limitera à la notion d'horloge mesurant l'écoulement du temps et permettant d'attendre une durée.

Par contre, contrairement aux systèmes considérés usuellement en automatique et en physique, une spécificité de la musique est de considérer des horloges autonomes et indépendantes et de mélanger un temps événementiel discret avec des notions continues. Nous reviendrons sur cette particularité essentielle plus tard.

En dehors de ces notions, les systèmes continus interviennent assez peu dans notre travail.

1. Par exemple, l'exécution d'instructions assembleur peuvent se recouvrir dans une exécution pipeliner alors que pour le programmeur les instructions se succède les unes à la suite des autres.

### 3.1.2 *Les systèmes événementiels*

Nous définissons ici un système événementiel comme un système dont l'organisation temporelle est décrite par des événements instantanés qui sont en relation de succession et de simultanéité. On peut définir ces relations « en soi », ou bien à travers une notion de date. Par exemple, la naissance de Wolfgang Amadeus Mozart s'est produite avant sa mort et cette relation de succession est logique et n'implique pas de dates. On peut aussi dire que Mozart est né en 1756 et qu'il est mort en 1791 et en déduire que la naissance de Mozart précède sa mort car l'année 1791 succède à 1756.

Nous analyserons différents modèles temporels mis en œuvre dans des langages de programmation suivant qu'ils mettent l'accent sur :

- la relation de succession (présentée à la section suivante) ;
- la relation de simultanéité (section 3.3) ;
- ou bien sur la datation des événements (section 3.4).

## 3.2 SUCCESSION

La relation de succession est une relation d'ordre (i.e. une relation antisymétrique et transitive). Elle peut être totale ou partielle. Du point de vue des langages de programmation, cette relation peut être explicite ou implicite, ce qui nous amène à classer les langages en trois catégories : les langages déclaratifs, les langages séquentiels et les langages parallèles ou concurrents.

LANGAGES DÉCLARATIFS. Dans les modèles fonctionnels et déclaratifs l'ordre de calcul est uniquement dicté par la causalité ; il n'existe pas de compteur d'instruction.

En 1974, Gilles Kahn présente un modèle sémantique fonctionnel flot de données qui s'appuie sur des calculateurs distribués, communiquant au travers de files de communication de taille non bornée [Gil74]. Ce modèle qui prendra ensuite le nom de *réseaux de Kahn*, garantit le déterminisme des calculs sans contraindre un ordonnancement.

Les modèles flots de données sont adaptés aux applications où la structure des calculs à effectuer change peu souvent. L'exemple typique est un traitement itératif sur des données disponibles périodiquement. Le temps dans ces modèles n'est pas manipulé directement, il est une conséquence des calculs qui s'exécutent lorsque les données en entrée sont présentes. On peut cependant intégrer le temps dans ces modèles en contrôlant le nombre d'entrées ou de cycles par unité de temps.

D'autres modèles permettent de s'abstraire de l'ordonnancement effectif des calculs, tout en garantissant un résultat cohérent, par exemple les modèles asynchrones tels que le langage dynamique

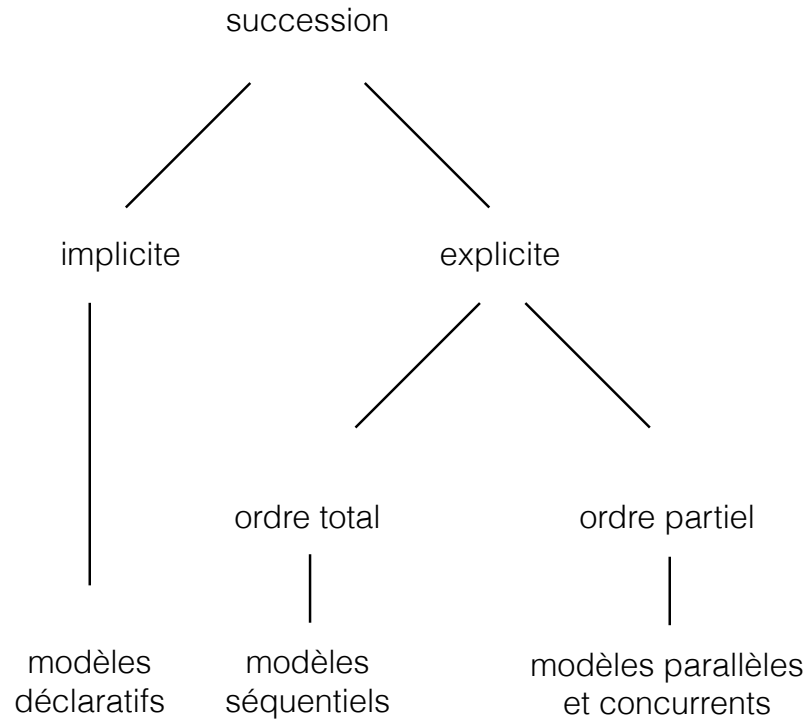



FIGURE 7: Notre classification des relations temporelles dans les langages de programmation.

Lynda centré sur la coordination de processus concurrents [CG89]. Son formalisme permet à l'utilisateur de raisonner sur la création des processus et leurs communications sans se soucier de leur mise en œuvre. La machine abstraite chimique (cham) [BB90] inspiré du langage Gamma [LM86, BLM90] est un autre exemple de modèle ou l'ordre des opérations n'a pas d'importance. Ce formalisme permet de modéliser simplement le non-déterminisme et a été utilisé pour spécifier la sémantique opérationnelle de diverses algèbres de processus et langages, comme par exemple le CCS, le $\pi$-calcul et les langages de coordination comme Linda. Il peut se décrire selon une métaphore chimique. Un programme est une solution chimique de données qui interagissent librement. Les données se comportent comme des molécules qui réagissent entre elles pour générer de nouvelles données, qui peuvent à leur tour réagir. Les réactions sont décrites par des règles de réécriture.

Le paradigme de programmation par contraintes est une autre forme de programmation déclarative. Là encore on ne spécifie pas une séquence d'étapes à exécuter mais plutôt les propriétés d'une solution à trouver. Le langage CCP [SR89] permet de modéliser un système concurrent où les calculs se dégagent de l'interaction d'agents d'exécution qui communiquent simultanément en définissant et en vérifiant des contraintes s'appliquant sur des variables partagées. Parmi

les extensions de ce modèle on peut citer NTCC [PV01] qui introduit la notion de temps vu comme une séquence d'unités temporelles. A chaque unité de temps un calcul CCP est lancé.

LANGAGES SÉQUENTIELS. Dans les modèles impératifs, les structures de contrôle sont utilisées uniquement pour spécifier la succession explicite des calculs. Les langages de programmation les plus connus tels que C ou Java reposent sur ce modèle de calcul. Il faut souligner que la relation de succession n'interdit pas nécessairement les relations de simultanéité. Par exemple, le langage C est séquentiel et ne peut exécuter qu'un seul calcul à la fois alors que le langage Esterel qui est aussi séquentiel, peut effectuer plusieurs actions dans le même instant logique (au bout du compte, elles s'exécuteront les unes après les autres mais la sémantique du langage permet de penser leur exécution *en même temps*).

LANGAGES CONCURRENTS OU PARALLÈLES. Alors qu'on a un compteur d'instruction pour les modèles séquentiels, on passe à $n$ compteurs d'instruction pour les modèles concurrents ou parallèles. Il s'agit des modèles de thread, de tâches, de processus, où chacune de ces entités a son propre compteur d'instruction indépendant. On différencie les modèles parallèles qui peuvent effectuer les calculs en même temps sur des ressources différentes des modèles concurrents où les calculs se partagent une ressource pouvant introduire du non-déterminisme.

## 3.3 SIMULTANÉITÉ

Quand deux événements ne se succèdent pas c'est qu'ils sont simultanés. Cette notion peut être plus ou moins « idéale », ce qui nous servira de grille de lecture.

### 3.3.1 *La simultanéité de durée zéro*

Pour faciliter les raisonnements temporels, le modèle synchrone met en place l'abstraction synchrone. Dans ce modèle, on suppose que les calculs et les communications peuvent s'effectuer dans le même instant et prennent un temps logique nul. La succession de ces instants définit un temps logique où seul l'ordre importe et non la durée entre deux instants.

Ce formalisme permet de spécifier des programmes indépendamment de toute architecture et de contrainte de la ressource temporelle. En découplant les problématiques temporelles sémantiques dues à l'algorithmique des contraintes temporelles opérationnelles dues à l'implémentation sur architecture particulière, cette approche a mon-

tré son efficacité pour la conception et la vérification d'applications critiques temps réel [BB91].

En effet, le temps a historiquement été considéré comme une contrainte opérationnelle. Il s'agissait de faire en sorte que chaque tâche ait terminé avant sa date limite ; le résultat des tâches étant disponible dès que l'exécution est terminée. Cette approche bénéficie de toute la théorie de l'ordonnancement temps réel [But97]. Cependant dans les applications temps réel, pour obtenir des résultats satisfaisants, les systèmes doivent réagir au bon moment, ce qui est très différent que de réagir le plus vite possible. Le temps doit donc être considéré comme une propriété sémantique.

Nous évoquerons trois exemples de langages qui suivent l'hypothèse synchrone.

LUSTRE. Lustre est un langage *flots de données* avec une vision fonctionnelle du calcul, adaptée aux applications où les calculs effectués changent peu souvent. Les signaux sont définis de manière causale, leur valeur pouvant évoluer suivant des hiérarchies temporelles complexes d'horloges imbriquées.

ESTEREL. *Esterel* [BG92] est un langage impératif et concurrent. L'exécution d'un programme progresse par étape. Une étape correspond à un instant logique. À chaque étape, le programme calcule ses sorties et son état en fonction des entrées et de l'état précédent. Les programmes avec des cycles de causalité sont refusés. Chaque tâche concurrente *Esterel* s'exécute au même rythme et communique à travers un mécanisme de signaux. À un instant donné, un signal est soit présent et on peut accéder à sa valeur instantanée, soit absent. Dans un cycle, les tâches qui lisent la valeur d'un signal attendent que les autres tâches aient fixées la valeur du signal : c'est la relation de causalité qui sert à ordonner les tâches dans l'instant. Des méthodes de propagation causale de *certitudes* sur la présence ou l'absence de signaux permettent de déduire l'état des autres signaux.

REACTIVEML. *ReactiveML* [MP08] est une extension réactive du langage Ocaml. Il est adapté aux applications où des processus concurrents communiquent entre eux et sont créés dynamiquement. La concurrence est basée sur un modèle synchrone où le temps est une succession d'instants partagés par l'ensemble des processus.

### 3.3.2 *La simultanéité « bornée »*

L'hypothèse synchrone d'actions effectuées instantanément, est un modèle idéal à deux titres :

– même si les actions se produisent en même temps, un ordre d'exécution correspondant à des contraintes de causalité doit être

respecté (par exemple, la valeur d'une variable doit être produite avant d'être utilisée) ;
– la réalisation physique d'une action prend un temps incompressible.

Pour répondre à ces deux problèmes, le modèle idéal synchrone peut s'affiner de plusieurs manières. Le modèle de temps superdense répond principalement au premier problème soulevé : comment concilier instant de durée zéro et succession. Les approches de type *temps d'exécution logique* répondent à la seconde question.

TEMPS SUPERDENSE. La notion de temps superdense [MP92, MMP91, CLL+06, LZ05] est un modèle temporel permettant de gérer ce que Gérard Berry appelle « l'épaisseur de l'instant ». Une valeur de temps superdense est un couple $(t, n)$ où $t$ correspond à une date et $n$ correspond à un index, aussi appelé micro-pas. Ainsi deux événements de date $(t1, n1)$ et $(t2, n2)$ pourront se produire l'un après l'autre ($n1 \neq n2$) bien qu'ils soient "physiquement" simultanés ($t1 = t2$).

MODÈLE DE TEMPS BASÉ SUR L'ANALYSE NON-STANDARD. Les systèmes hybrides sont des systèmes dynamiques faisant intervenir des modèles continus et événementiels. Afin de résoudre les problèmes rencontrés aux frontières entre le discret et le continu, plusieurs modèles de temps [BBCP12, BF14, MSZ] se fondent sur l'analyse non-standard [Cut88]. Chaque date temporelle réelle est composée d'une infinité de successeurs et prédécesseur séparés d'une valeur infinitésimale sans progression du temps réel. Ce modèle permet de formaliser des comportements non désirés comme ceux liés au paradoxe de Zénon (insérer une infinité d'événements discrets entre deux dates réelles).

Selon Edward Lee, le fait de devoir adapter les méthodes d'analyse non standards afin de pouvoir définir une sémantique opérationnelle impliquant la notion d'étapes discrètes est équivalent avec la notion plus simple de temps superdense [Lee14].

LE MODÈLE DE TEMPS D'EXÉCUTION LOGIQUE (LET). Le modèle de *Temps d'Exécution Logique* (LET) [Kop91] est un compromis entre l'approche purement synchrone, temps d'exécution zéro (ZET) et une approche classique des systèmes temps réel où les entrées-sorties peuvent se faire pendant toute la durée de la tâche, temps d'exécution limité (BET). L'approche LET est à priori moins efficace qu'une approche BET du point de vue de l'utilisation des ressources, mais elle est plus robuste. Elle est également plus proche des réalités computationnelles d'une approche théorique purement synchrone.

*Giotto* est le premier langage supportant le modèle LET. Ce langage permet la spécification d'applications pour les systèmes embarqués avec des contraintes temps réel fortes. L'idée de ce langage et de

ses successeurs [PT08, GSVK+06] est de faciliter le découplage entre la spécification fonctionnelle et l'architecture du sytème. La spécification d'une tâche est capturée par un événement déclencheur et un événement de terminaison tandis que le temps d'exécution effectif est obtenu par analyse en fonction de l'architecture. L'événement déclencheur lance l'exécution de la tâche tandis que l'événement de terminaison libère les sorties de la tâche. Bien que l'exécution de la tâche puisse se terminer avant cet événement, les sorties sont disponibles seulement au moment de l'événement de terminaison. Le temps logiciel est assumé comme étant le même que le temps réel. Ce modèle permet d'exécuter les programmes de manière déterministe. En séparant les considérations temporelles des considérations fonctionnelles il facilite la réalisation et la portabilité des programmes. Alors que *Giotto* est purement « time-triggered », *xGiotto* gère aussi les événements asynchrones, il est donc également « event-triggered ». Ce langage introduit un mécanisme de portée pour la gestion des événements.

### 3.3.3 *L'absence de simultanéité*

Une approche très répandue est de considérer que deux actions ne peuvent pas avoir lieu en même temps : c'est l'approche asynchrone. Dans cette approche, c'est la succession qui doit être définie par l'utilisateur mais celle-ci peut être partielle : la construction PAR en Occam permet par exemple de spécifier que deux actions ne sont pas en relation déterminée de succession. Elles peuvent donc se dérouler en même temps, mais c'est une conséquence de l'indéterminisme et non une attente du modèle. Un exemple typique de cette catégorie de langage est ADA.

ADA. Ada [ada97] est un langage de programmation impératif structuré, statiquement typé, qui possède des instructions permettant de gérer la notion de tâche concurrente et de rendez-vous entre tâches, largement utilisé dans le domaine des systèmes temps réel et embarqués. Bien qu'il offre des opérateurs temporels de délai ou de datation, Ada n'offre pas des garanties fortes sur les comportements temporels. Par exemple l'instruction wait 4 ; wait 5 n'est pas équivalente à wait 9.

LES EXTENSIONS TEMPS RÉEL DES LANGAGES CLASSIQUES. L'approche mise en œuvre dans *Ada* est caractéristique de toute une série d'extension de langages classiques vers le temps réel. On peut citer Real-time Java ou encore Real-Time Posix qui sont utilisés surtout pour des applications avec des contraintes temps réel souples [BW01]. Ces extensions reposent sur l'ajout de fonctionnalités concernant la gestion de la mémoire, les threads, les mécanismes de synchro-

nisation et de communication, réduisant ainsi les incertitudes temporelles.

## 3.4 LA DATATION DES ÉVÉNEMENTS

La problématique du temps-réel ne se réduit pas à effectuer les calculs dans le bon ordre, mais à assurer qu'ils sont réalisés au bon moment. Notons que pour cela, il ne suffit pas de les effectuer suffisamment vite : une note de musique doit par exemple être produite « au bon moment » et pas « avant telle date ».

Les modèles asynchrones sont particulièrement désavantagés pour répondre à ce type de contrainte. Par exemple, il n'est pas possible de réaliser l'action « à midi, faire ceci » puisque par définition, les calculs ne peuvent avoir lieu simultanément et donc ne peuvent avoir lieux à une date précise. Pourtant, l'approche asynchrone est celle qui sous-tend les extensions des langages classiques vers le temps-réel.

Les langages synchrones sont a priori plus armé pour répondre aux problèmes posés par le temps-réel, grâce justement à un traitement bien fondé de la simultanéité. Cependant, pour pouvoir spécifier « à midi, faire ceci », il faut qu'une notion de date, par exemple « midi », existe dans le langage. Or dans les exemples précédents la notion de dates n'existe pas en dehors des événements. La notion de date ou de durée (quantité de temps entre deux événements) est généralement repoussée dans l'environnement : on suppose qu'un service « extérieur » produit un événement qui correspond à la date « midi » et avec lequel on pourra se synchroniser.

Sans l'existence de ces événements extérieurs, il n'est pas possible de dater les événements : la quantité de temps qui s'écoule entre deux événements n'est pas une quantité connaissable dans le langage et elle ne peut pas influencer le calcul.

Rien n'empêche pourtant de dater les événements, dans un modèle synchrone, par exemple avec une date $t \in \mathbf{Q}$. Il est ainsi toujours possible de considérer un événement prenant place entre deux événements arbitraires. Nous appellerons ces modèles *modèles à événements discrets*.

MODÈLE À ÉVÉNEMENTS DISCRETS Le modèle à événements discrets repose sur des interactions discrètes et temporisées entre les composants du modèle. Une interaction est un événement daté considéré comme instantané, chaque acteur réagissant aux événements en entrée selon un ordre bien défini. Le temps dans ces modèles avance avec la séquence des événements qui peut être associée au temps réel, mais ce n'est pas toujours le cas. Un des des plus anciens formalismes à événements discrets est Discrete Event System Specification (DEVS)[KZ87]. Ce modèle a inspiré les langage de description de matériel tels que SystemC, VHDL ou Verilog. Le sytème hétérogène Pto-

lemy et le modèle de programmation PTIDES supportent ce modèle à événement discret basée sur le modèle de temps superdense. Lee définit une sémantique du modèle à événements discrets comme une généralisation du modèle synchrone [LZ07].

Ce modèle est utilisé dans la conception de *Act* [MS14], un langage de programmation de haut niveau pour les systèmes temporisés orientés acteur. Le but est de proposer un langage où le temps est réellement considéré comme un citoyen première classe tout en garantissant un comportement déterministe. Il s'agit d'une notion du temps logique partagé par tous les acteurs du réseau. Le temps logique peut être associé à du temps physique pour la programmation de systèmes temps réels, selon les principes appliqués dans PTIDES [ELM+12, ZLL07].

## 3.5 COMBINER PLUSIEURS TEMPS

Le temps dans les modèles de calcul est parfois considéré à plusieurs échelles. D'autres fois des modèles temporels de natures différentes doivent interagir. Nous présentons dans cette section certaines approches qui permettent la cohabitation de multiples modèles de temps. Dans la suite de ce manuscrit nous désignerons la cohabitation de plusieurs modèles de temps au sein d'un même système par le terme de *temps multiples*.

Nous évoquons ci-dessous plusieurs domaines applicatifs qui nécessitent la prise en compte de temps multiples et des recherches qui ont abordé cette question.

PLUSIEURS MANIÈRES DE COMPTER LE TEMPS. Il y a plusieurs manières de voir le temps passer, un temps pouvant être à la fois physique (la date d'une horloge autonome et extérieur au système), logique (l'occurrence d'événements) ou hiérarchique (des échelles imbriquées des deux types de temps précédents). La notion de temps multiforme caractérise un temps pouvant être à la fois physique, logique ou hiérarchique. Plusieurs modèles et langages disent pouvoir gérer des systèmes combinant ces différents temps. On citera par exemple le langage synchrone Esterel [BG92] qui utilise le terme de temps multiforme et qui, à travers son modèle de temps événementiel, permet de manipuler indifféremment ces différents temps : il n'y a pas de différence fondamentale entre attendre 10 mètres et attendre 10 secondes quand on va à la vitesse de 1 $m.s^{-1}$.

Marte, extension du modèle UML pour le domaine du temps réel embarqué permet la spécification de modèles où à coté du temps physique, un temps logique peut être associé à des horloges définies par l'utilisateur [AMdS07]. Les durées peuvent par exemple être comptées en nombre de pas d'exécution ou de cycles d'horloge.

PTOLEMY. Ptolemy [Pto14] est un outil de simulation qui permet également de manipuler un temps logique et hiérarchique. Chaque composant, quelque soit le modèle de calcul (synchrone, événements discrets, etc.), maintient une horloge locale pouvant évoluer indépendamment du temps global de l'environnement. Des mécanismes de communication et de synchronisation adéquats permettent ensuite de faire cohabiter ces temps locaux. Ptolemy II étend l'approche précédente pour permettre des interactions entre plusieurs modèles hétérogènes de temps à travers la communication entre acteurs. À chaque acteur correspond un modèle de temps particulier et la communication entre les acteurs de différentes natures est définie formellement. La figure 8 montre les différents modèles de calculs qui peuvent interagir. Ces modèles et leurs interactions sont formellement bien définis.

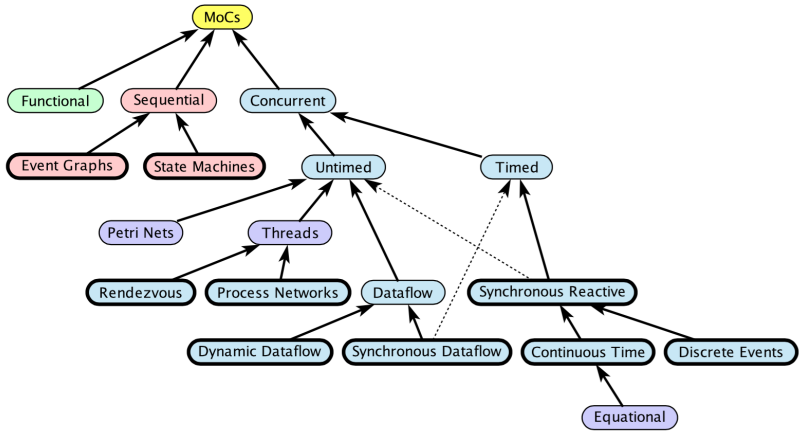



FIGURE 8: Relations entre les modèles de calculs implémentés dans Ptolemy II (figure extraite de [Pto14])

AUTOMATES TEMPORISÉS. Les systèmes réactifs obéissent le plus souvent à des contraintes temporelles qui admettent un ensemble d'exécutions possibles. Les automates temporisés constituent un modèle de systèmes réactifs à temps continu permettant de spécifier ce type d'ensembles [AD94, AHV93]. Ils offrent la manipulation des aspects temporisés (délais, durées) de manière explicite et peuvent être analysés grâce à des outils de vérification formelle.

SYSTÈMES CYBER-PHYSIQUES. Un système cyber-physique désigne l'orchestration de systèmes informatiques interagissant avec des systèmes physiques tels que des processus mécaniques, chimiques, etc., et de contrôle en boucle fermée comme étudiés en automatique [Lee08]. Les calculs dépendent des processus physiques et vice versa. Ce sont des systèmes hétérogènes, c'est-à-dire des systèmes dans lesquels différents modèles de calculs cohabitent, tels des systèmes d'équa-

tions différentielles à temps continu qui modélisent des systèmes physiques ou des modèles synchrones à temps discret souvent utilisés dans les applications temps réel. Ces différents modèles sont difficiles à combiner ce qui justifie l'émergence de la discipline des systèmes cyber-physiques.

*L'humain dans la boucle de de rétroaction*

Souvent les systèmes cyber-physiques se déploient dans des environnements où l'interaction avec un humain joue un rôle clé (santé, énergie, environnement, transports, musique) De plus en plus d'études tendent à expliciter l'humain au centre de ces modèles, en le considérant comme un opérateur, un perturbateur ou une source de données de la boucle de rétroaction. Le comportement humain peut être partiellement modélisé en s'appuyant sur des techniques en sciences cognitives, en apprentissage automatique, sur l'études des réseaux sociaux, etc.

SYSTÈMES GALS Les systèmes globalement asynchrones, localement synchrones (GALS) sont des systèmes composés de sous-systèmes synchrones, interagissant de manière asynchrone. Différentes solutions ont été proposées pour gérer les problèmes qui apparaissent aux frontières entre les deux modèles : des extensions des modèles synchrones [BS01], des extensions des modèles asynchrones [DMK+06], des langage dédiés [JLM14], etc.

## 3.6 SYNCHRONISATION

Dans les sections précédentes, nous avons présenté un panorama des modèles de temps en informatique, et des langages de programmation qui les supportent à travers des notions de succession, de simultanéité et de durée.

Ces thématiques peuvent s'étendre à la problématique plus large de la synchronisation entre des systèmes conçus comme autonomes. En électronique, en automatique, en traitement du signal et dans les autres domaines où le temps joue un rôle clé, différents mécanismes ont été développés afin de permettre à deux systèmes de partager une notion de temps commune ou simplement de synchroniser deux événements. Nous mentionnons quelques approches qui abordent cette question.

CONSTRUCTION DE LA CAUSALITÉ DANS UN SYSTÈME DISTRIBUÉ En 1978 Lamport décrit un algorithme permettant de déterminer l'ordre des événements dans un système distribué [Lam78]. Il en déduit moyen pour synchroniser les horloges physiques de processus distribués. Cet algorithme peut être vu comme une manière de contrôler le modèle asynchrone au cours du temps.

SYNCHRONISATION EN AUTOMATIQUE En automatique les problèmes de synchronisation sont résolus par des techniques de régulation et d'asservissement [DB94]. L'approche la plus naïve pour coordonner l'évolution de deux paramètres consiste à enclencher les dispositifs qui les contrôlent de manière synchrone afin de les faire évoluer en parallèle. Il s'agit de systèmes d'asservissement à boucle ouverte qui ne peuvent pas s'adapter à d'éventuels changements au cours de l'exécution (fréquence d'échantillonnage, décalage temporel).

Un système asservi à boucle fermé est un système commandé possédant un dispositif de retour permettant de contrôler la valeur d'un paramètre en limitant l'écart par rapport à sa valeur de consigne quelles que soient les perturbations externes. En particulier la boucle à phase asservie, en anglais « Phase-Locked Loop » (PLL), permet de synchroniser deux signaux, et plus précisément de générer un signal de sortie avec la même fréquence et la même phase qu'un signal de référence [Vit63].

PROTOCOLES NTP ET PTP Une technique directe pour partager une notion de temps commune est de diffuser une horloge à travers un réseau de communication. Cette technique est utilisée dans le système GPS (*Global Positioning System*) et permet la synchronisation d'horloges à $10^{-7}$ secondes près.

Les protocoles de communication *Network Time Protocol* (NTP) [Mil91] et *Precision Time Protocol* (PTP) [ptp05] permettent de maintenir un réseau d'horloges synchronisés en échangeant des messages asynchrones datés qui permettent de corriger les éventuelles dérives des horloges locales. Ces protocoles n'ont pas vocation à déterminer un horodatage, mais à le transmettre. La précision des techniques utilisés dans ces protocoles ne dépendent pas du temps de communication mais uniquement de l'asymétrie entre les délais de communication. Si la latence de communication entre un point A et un point B est la même que celle du point B au point A alors les horloges seront parfaitement synchronisées.

Qu'il s'agisse des travaux de Lamport, ou bien des protocoles *Network Time Protocol* (NTP) et *Precision Time Protocol* (PTP) l'objectif est de réajuster des horloges locales qui évoluent approximativement à la même vitesse.

SYNCHRONISATION DANS LES SYSTÈMES MULTIMEDIAS. Les travaux dans le domaine des systèmes multimedias s'étendent des bases de données, aux sciences cognitives en passant par la communication dans les réseaux à grande échelle [SN04]. La multiplication des applications de plus en plus hétérogènes dans ce domaine a amené à une complexité croissante dans les dépendances temporelles entre les différents types de médias. Dans certaines de ces applications, des

medias de différentes natures (continus ou discrets), dont les données peuvent arriver à des débits et des taux d'échantillonnages variables, doivent se synchroniser pour leur présentation. Ces systèmes se concentrent davantage sur la qualité de service plutôt que sur le temps des calculs.

Semantic Time Framework [Lee07] est un projet autour des systèmes multimedias interactifs. Le but est de créer un ensemble d'outils théoriques et logiciels pour simplifier le développement d'applications multimédia, en particulier les applications qui permettent à l'utilisateur de manipuler l'évolution temporelle d'un fichier audio ou vidéo. Ces outils reposent sur un formalisme appelé *semantic time*, pour la représentation du temps et des transformations temporelles. Ce formalisme définit des intervalles sémantiques de temps inspirés des intervalles de Allen et des pulsations musicales. Des fonctions du temps décrivent la relation entre ces intervalles et le *temps de présentation*, et permettent la synchronisation fine d'un flux audio, ou vidéo avec un signal d'entrée. Par exemple, au lieu de jouer sur la position du flux de sortie par rapport au flux d'entrée pour les resynchroniser en cas de perturbation, c'est la vitesse qui est modifiée pour aligner les deux flux à une certaine date dans le futur, sans discontinuité dans le flux de sortie.

## 3.7 POSITIONNEMENT D'ANTESCOFO

Dans cette dernière section nous positionnons *Antescofo* vis à vis des notions que nous venons d'introduire.

ANTESCOFO, UN SYSTÈME CYBER-PHYSIQUE. Les œuvres en musique interactive sont des systèmes cyber-physiques où le musicien peut être vu comme un composant à part entière de la boucle de rétroaction. Le langage d'*Antescofo* permet au compositeur de décrire un tel système dans lequel coexistent le temps de la composition, de la performance et les temps propres à chaque processus musicaux étant eux-même de natures hétérogènes.

UN LANGAGE. Le langage d'*Antescofo* partage l'idée de simultanéité introduit par les langages synchrones. On suppose que les actions atomiques d'Antescofo prennent un temps nul. Comme dans *Lustre* [HCRP91], *Signal* [LGLBLM91] et *Lucid Synchrone* [Pou06], le programmeur peut accéder aux valeurs antérieures des variables. Mais contrairement à ces langages il peut y accéder de différentes manières (logique ou chronométrique). De plus, le style de programmation d'*Antescofo* n'est pas déclaratif, il se rapproche plus du style impératif d'*Esterel* [BG92]. *Antescofo* se distingue aussi par son caractère dynamique : on peut créer des processus en parallèle et à la volée. *ReactiveML* [MP08] permet aussi la création de processus dynamiques

mais dans un modèle temporel où il faut gérer de manière explicite la notion d'instant courant et le passage d'un instant à une autre. Le langage *Antescofo* est également proche du langage *Act* [MS14] dans sa conception puisqu'il permet de gérer la durée dans un modèle synchrone. Cependant dans *Antescofo* la durée est manipulée par l'utilisateur à travers le tempo du musicien et la spécification de temps multiples.

DES STRATÉGIES DE SYNCHRONISATION. Les techniques développées dans le langage d'*Antescofo* pour la synchronisation des actions électroniques avec un environnement musical reposent à la fois sur des idées de rendez-vous développées dans les modèles asynchrones discrets, par exemple dans un langage comme ADA (voir par exemple la notion de stratégie `@tight` chapitre 6), et aussi sur des techniques de synchronisation continues que l'on retrouve dans les systèmes asservis en automatique. De plus, ces dernières techniques sont aussi au fondement de l'algorithme de suivi de tempo utilisé dans *Antescofo*.

UN MODÈLE DE TEMPS MULTIPLE. Pour *Antescofo* les relations temporelles sont des entités explicites du langage, faisant de celles-ci des propriétés sémantiques et non une propriété opérationnelle de l'implémentation. Cette approche est défendue par Gérard Berry et Edward Lee, entre autres. Ce dernier propose six caractéristiques qui doivent être intégrées dans un langage de haut-niveau afin que le temps soit considéré comme une entité de première classe dans les systèmes distribués de programmation en temps réel : permettre l'expression des contraintes temporelles, offrir des mécanismes de communication en temps réel, assurer des contraintes temporelles, gérer le non-respect de ces contraintes, assurer la cohérence distribuée de l'état du système dans le domaine temporel, et permettre la vérification statique des relations temporelles [LDFW87].

La relation musicien-machine peut être vue comme un système distribué, même s'il n'est pas conçu comme tel au départ. On peut donc aborder le langage à travers les six caractéristiques précédentes. Le langage permet de gérer les contraintes temporels musicales ou physiques grâce à des structures de contrôles adaptées supportant la notion de tempos multiples. Grâce notamment aux différentes stratégies de synchronisation et à la gestion des erreurs du musicien, l'utilisateur peut spécifier simplement les réactions attendues de la machine, par rapport aux comportements du musicien qui ne sont connus que pendant la performance. Pour ce qui est des vérifications statiques, le caractère dynamique d' *Antescofo* limite ce qui peut être analysé. Cependant une partition peut faire l'objet d'analyse statique, ce qui a déjà été fait dans [FJ13, PJ15].

# 4

# LES USAGES DU TEMPS EN INFORMATIQUE MUSICALE

L'informatique musicale est riche de langages dédiés [BHN99, CLRC05] aux diverses activités que l'on rencontre à chacune des étapes du processus de création musicale. Dans les sections suivantes, nous examinerons plusieurs exemples de langages dédiés classés selon la notion de temps mis en jeu. Nous avons répertorié trois notions temporelles distinctes qui interviennent dans les processus musicaux :

- le temps audio (section 4.1),
- le temps comme une donnée de calcul dans la composition (section 4.2),
- le temps performatif du concert (section 4.3).

La réalisation de ces temps dans un système informatique pose la question de leurs relations et de leur mise en œuvre dans le temps physique. Cette question sera abordée sous l'angle des séquences, qu'elles soient audio ou symbolique (section 4.4) et de structures plus complexes correspondant à des scénarios musicaux qu'ils soient écrits à l'avance ou improvisés (section 4.5).

## 4.1 LE TEMPS AUDIO-NUMÉRIQUE

Dans les ordinateurs le son est stocké sous la forme d'une suite de nombres correspondant à la valeur de l'amplitude à un instant donné selon une fréquence d'échantillonnage et une précision en nombre de bits. Beaucoup de langages de programmation en informatique se focalisent sur le traitement, l'analyse et la synthèse des sons à travers le calcul audio numérique. Ils peuvent être destinés à des applications en temps réel ou en temps différé. Nous verrons dans une autre section les problématiques supplémentaires induites par le cas du temps réel. Ces langages supportent la plupart des modèles de synthèse qui ont été développé tout au long de ces 50 dernières années : synthèse par table d'onde, synthèse concaténative, synthèse additive, synthèse soustractive, transformée de Fourrier, etc [Bre07]. Les langages de synthèse donnent la possibilité au compositeur de créer lui-même ses processus de synthèse sonore. Le temps manipulé ici correspond au temps du signal numérique.

Nous présentons trois langages dédiés à la synthèse sonore. Les autres langages seront présentés dans les sections suivante, sous des points de vue différents, bien qu'ils supportent aussi le calcul audio.

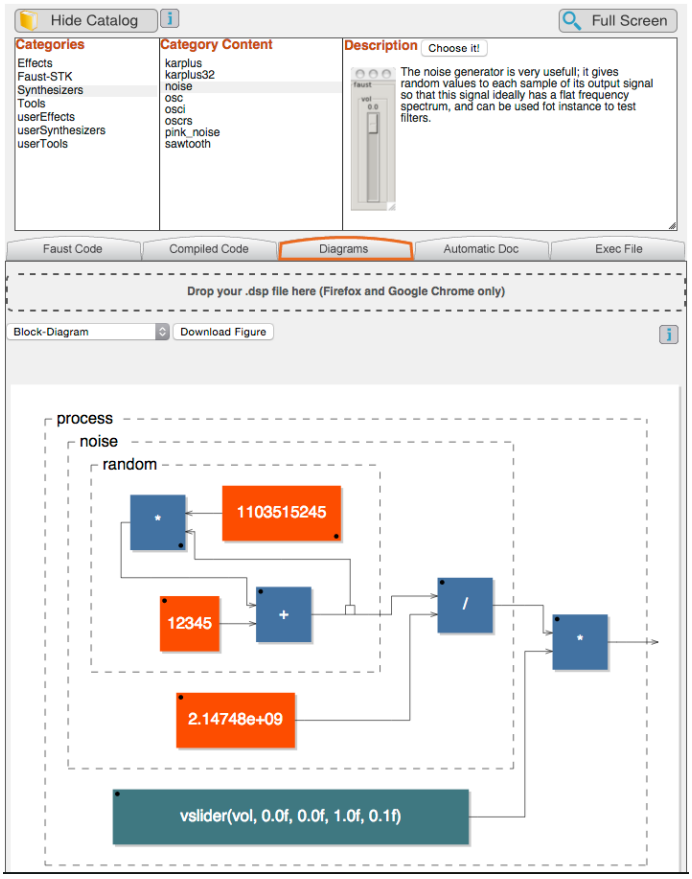



FIGURE 9: Bloc diagramme d'un générateur de bruit en Faust.

CSOUND Dans le domaine du traitement du signal et de la synthèse sonore, *CSound* [Bou00], héritier de la famille des langages *MusicN* [MMM+69], est basé sur un langage facilitant le développement de processus audio sous la forme de graphes flot de données, d'univers sonores personnels. On définit séparément l'orchestre, décrivant la nature des sons électroniques, et la partition, qui décrit des paramètres temporels contrôlant les instruments. Bien que des éléments pour le contrôle en temps réel soient intégrés dans les dernières versions, le langage de *CSound* ne bénéficie pas d'outils dynamiques de haut niveau. De ce fait, le couplage des processus de synthèse *Csound* avec un environnement musical incertain est difficile à mettre en place.

FAUST Faust [OFL09] propose un langage fonctionnel avec une sémantique bien fondée permettant de définir des modules de traitement de signal multi-plateformes et efficaces (cf. Figure 9). Le style est très similaire au langage FP proposé par Backus [Bac78] et s'inscrit donc dans le paradigme flux de données. Il est par exemple difficile de « rerouter » les signaux d'entrées entre des modules d'effets ou

de séquencer des traitements différents en fonction d'une condition logique arbitraire ; il n'est globalement pas adapté à la description temporelle de processus dynamiques ou bien de processus de haut niveau.

MODALYS Le logicel Modalys est un cas un peu à part puisqu'il ne s'agit plus de décrire le son directement mais les phénomènes physiques qui en sont la cause. C'est un logiciel de synthèse modale permettant de simuler les vibrations dans l'air, d'objets simples couplés entre eux (corde, plaque, etc) excités par d'autres objets (archet, marteaux, etc.). La spécification de ces systèmes acoustiques se fait à travers une interface en Lisp.

## 4.2 LE TEMPS RÉIFIÉ DE LA COMPOSITION

L'intégration de représentations symboliques dans le processus de composition n'est pas une idée qui est apparue avec l'informatique. Pourtant cette discipline a largement contribué au développement et à l'exploration de nouveaux types de représentions musicales. Le temps dans ces modèles est souvent vu comme une donnée manipulable et donc calculable à loisir. On parle alors de réification. Le temps réifié ne prend pas de valeur dans le temps physique ou réel jusqu'à ce que le processus décrit soit exécuté.

LES LANGAGES POUR LA COMPOSITION ASSISTÉE PAR ORDINATEUR. Différents paradigmes de programmation ont été utilisés pour les langages de composition assistée par ordinateur (CAO). Mais le paradigme de programmation dominant dans les langages de CAO est la programmation fonctionnelle. Les langages et environnement les plus connus sont OpenMusic [BAA09], Patchworks [Lau89] et Common Music [Tau91] tous les trois reposant sur le langage CommonLisp (cf. Figure 10).

L'approche interprétée n'est cependant pas une contrainte du domaine, comme l'exemple d'Euterpea le montre. Euterpea [Hud14] est le dernier né d'une famille de langages pour l'informatique musicale, qui reposent sur le langage fonctionnel Haskell. La représentation de séquences héritent des séquences paresseuses d'Haskell, ce qui permet d'exprimer simplement des algorithmes complexes. On retiendra en particulier les travaux sur les représentations polymorphes des medias temporels qui définissent une théorie algébrique [Hud04].

Les langages dédiés à la composition musicale assistée par ordinateur permettent aux compositeurs de construire leur propre environnement musical à travers la définition et la manipulation de processus fonctionnels en générant ou en transformant des données musicales. Ils ont historiquement été associés au temps différé de la composition, c'est-à-dire à la préparation du matériau sonore en amont de la per-

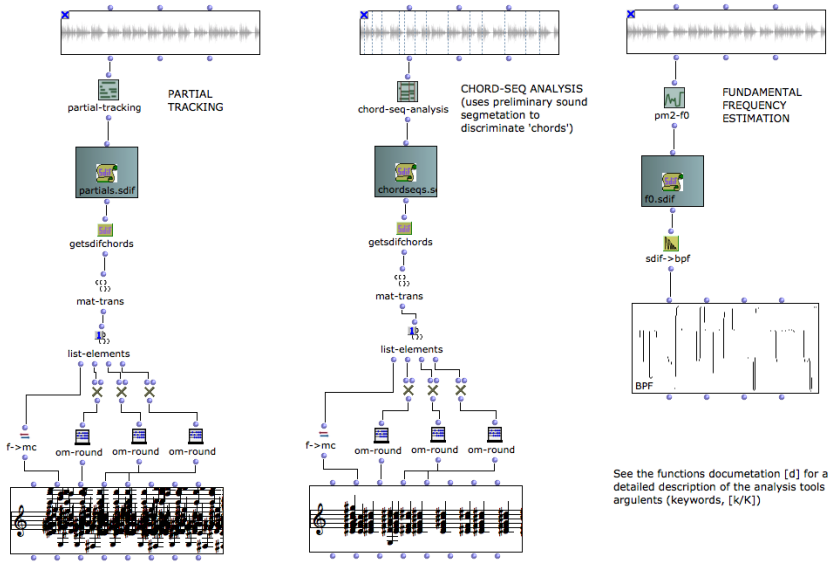



FIGURE 10: Exemple de patch OpenMusic manipulant à la fois des données symboliques et des données audio.

formance. Ils sont naturellement opposés aux langages dynamiques dédiés à la performance tels que Max [Puc91]. Ce qui différencie principalement le temps symbolique (réifié) de la composition et celui de l'exécution est le fait que le premier, peut aller dans toutes les directions du temps. Pourtant, les travaux récents dans OpenMusic pour modéliser la réactivité [BG14] ou le développement de l'outil de composition *Bach* [AG13] au sein de l'environnement temps réel Max, tendent à atténuer les différences entre logiciels pour la composition et logiciels pour la performance.

## 4.3 LE TEMPS PERFORMATIF

Le temps performatif est le temps du concert où le musicien-interprète joue la partition écrite par le compositeur en réalisant notamment les durées des événements qui y sont décrites. C'est également le temps où le musicien-improvisateur construit une interprétation selon une structure donnée ou seulement selon son humeur. Le temps performatif est aussi le temps pendant lequel des programmes informatiques, décrits par des langages de programmation, écoutent, s'exécutent et/ou se synchronisent avec leur environnement.

Nous proposons d'aborder ce temps performatif à travers des systèmes informatiques proches du langage *Antescofo*. Le premier est très ressemblant d'un point de vue conceptuel, les deux systèmes du deuxième point sont les environnements hôtes d'*Antescofo*, et enfin le

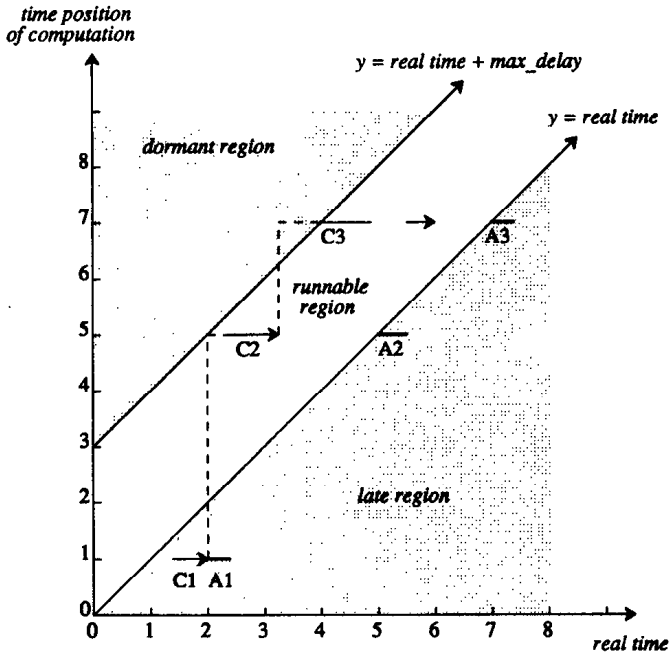



FIGURE 11: Exemple d'ordonnancement dans le logiciel *Formula* où les calculs sont exécutés dans une région temporelle de manière à anticiper les sorties sans dépasser (en arrière) le délai maximum autorisé. Figure extraite de [AK90]

troisième décrit une application pratique de notre système, l'accompagnement automatique.

FORMULA. Formula est un système temps réel pour la performance musicale développé à la fin des années 90 par Anderson et Kuivala [AK90]. Il permet de spécifier des programmes où des tâches de calculs et des messages envoyés vers des synthétiseurs sont ordonnancés en fonction de données issues d'un musicien interprète. On retrouve des notions intéressantes comme par exemple la possibilité de définir une fenêtre temporelle pour l'évaluation des processus permettant d'optimiser l'ordonnancement (cf. Figure 11), ou encore la possibilité de définir des mécanismes de déformation temporelle pouvant être appliqués à des séquences musicales. La gestion du temps est donc virtuelle mais la sémantique du langage associé n'intègre pas de relation particulière entre l'écoute et la réaction. En particulier, il n'y a pas de notions de tempo associé aux événements de l'environnement et les déformations temporelles sont définies par des formules analytiques connues à l'avance (autrement dit, on connait leur effet dans le futur).

L'ABSENCE DE TEMPS DANS MAX ET PUREDATA. *Max* et *PureData* [Puc91] sont deux des langages temps réel les plus utilisés dans

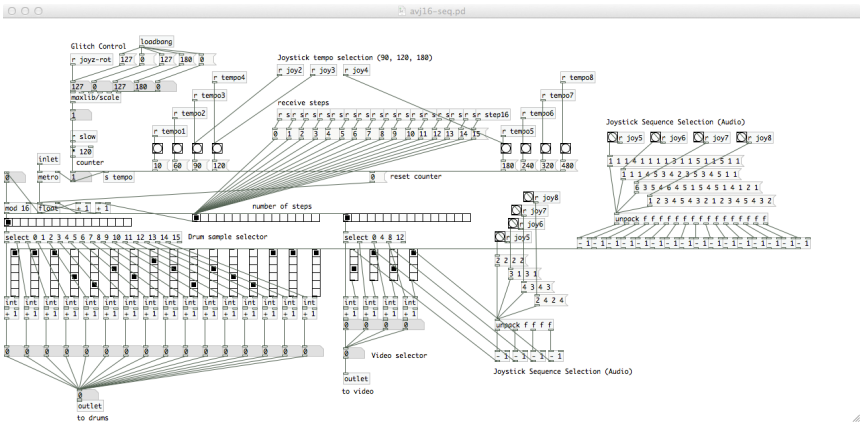



FIGURE 12: Séquenceur programmé dans l'environnement graphique Pure Data.

la communauté de la création multimédia (cf. Figure 12) Ce sont des langages visuels à flots de données et adaptés à la construction de modules hiérarchiques de traitements de signal et de contrôle, facilement connectables entre eux. Ils permettent de définir de manière intuitive des comportements fonctionnels, mais il est plus fastidieux d'organiser et de contrôler les changements d'états d'un scénario temporel complexe. Il faut noter que la difficulté à répondre à certaines classes de problèmes (en particulier les traitements dont la structure est dynamique) dans un contexte *flux de données déclaratif* a aussi été relevée dans le domaine des langages synchrones où il apparait clairement qu'une approche impérative comme celle développée par exemple par Esterel est complémentaire des approches déclaratives comme celle développée par Lustre et répond à des besoins spécifiques d'autres types d'applications (contrôleurs versus traitement du signal par exemple).

ACCOMPAGNEMENT AUTOMATIQUE. Roger Dannenberg, co-inventeur du suivi du partition, a proposé tout au long de sa carrière différentes solutions pour l'accompagnement automatique. En plus de son système originel [Dan84a], il a notamment développé un système d'accompagnement automatique avec plusieurs entrées qui à partir d'un ensemble de règles permet de déduire la position dans la partition [GD94]. Plus récemment, il a proposé un environnement pour la coordination de différents medias [LXD11] permettant d'adapter leur temporalité en tenant compte des propriétés (latence, fréquences d'échantillonnage, etc.) spécifiques à chacun de ces medias. Les techniques employées permettent une synchronisation fluide en jouant sur le tempo pour recaler la position de l'accompagnement en fonction des données en entrée (cf. Figure 13). Bien qu'il ait introduit de nombreux mécanismes pertinent pour la coordination musicale de

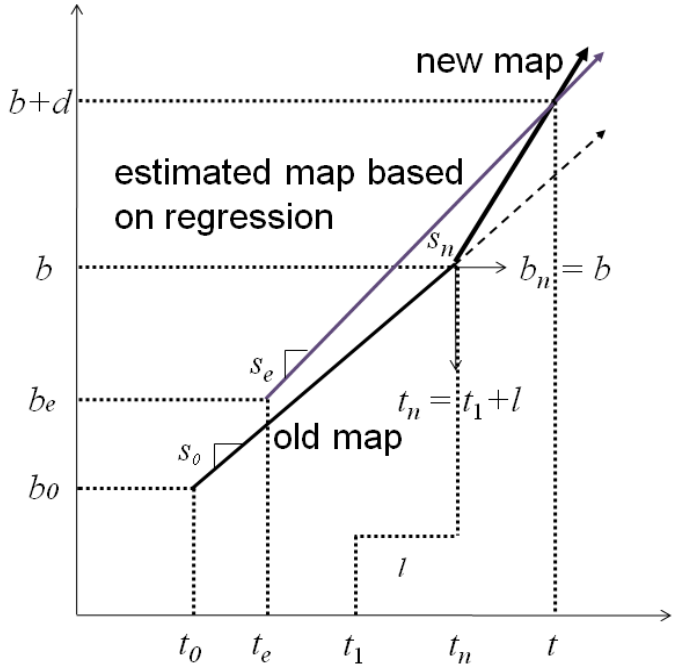



FIGURE 13: Figure extraite de [LXD11]. Ce diagramme représente le *mapping* entre le temps absolu et la position dans un référentiel correspondant à une partition. Le tempo du media contrôlé est adapté pour coller à l'estimation du temps global en tenant compte de la latence propre au media.

programmes électroniques, les outils développés ne sont pas adaptés à la spécification de scénarios interactifs plus complexes.

Citons également les travaux de thèse d'Andrew Robertson qui étudie la synchronisation d'un accompagnement (un fichier audio) à partir d'informations issues du jeu d'un batteur et analysées via un suiveur de pulsations [Rob09].

## 4.4 JOUER UNE SÉQUENCE DANS LE TEMPS

Dans cette section et la suivante on s'intéresse aux relations temps audio, temps réifié et temps de la performance. On examine d'abord comment on peut jouer une séquence audio (correspondant à la représentation en mémoire d'un signal physique) ou une séquence symbolique. Jouer une partition MIDI demande à passer d'un référentiel logique (une note est après une autre) à un référentiel physique (à quelle date en seconde je dois jouer une note).

### 4.4.1 *Calcul audio en temps réel*

Il est trop coûteux et trop difficile de produire un échantillon audio à la date exacte à laquelle il doit être restitué. L'approche habituelle est donc d'agréger les calculs audio dans un bloc d'échantillons, ce qui présente deux bénéfices. Le premier est que le calcul sur une séquence de $n$ échantillons est moins couteux que $n$ calcul de 1 échantillon (effet pipeline, vectorisation des accès mémoire, parallélisation de boucle, minimisation des interruption, etc). Le second bénéfice est que la contrainte temporelle a été relâchée : au lieu de devoir produire un échantillon à une date précise $n$ fois, il suffit de produire un bloc d'échantillon avant une date précise. La fréquence de calcul correspond généralement à la fréquence d'échantillonnage divisée par la taille du bloc de façon à ce que que le temps calculé soit le même que le temps réel.

L'utilisation de blocs dans le traitement audio introduit aussi de nouvelles problématiques : comment prendre en compte un contrôle qui intervient n'importe quand dans un traitement qui est structuré par blocs. Ces problèmes apparaissent dans des systèmes comme Max, PureData ou SuperCollider, qui permettent de faire dépendre des calculs audio de calculs de contrôle arbitraires (par exemple une interaction avec l'environnement). Ces systèmes doivent résoudre des problématiques de synchronisation entre les tampons (doit-on retarder un bloc pour le synchroniser avec une fréquence de calcul globale ?) mais aussi de coordination entre les données de contrôle et les données audio (doit-on considérer uniquement la dernière valeur de la donnée de contrôle dans le calcul audio ou doit-on prendre en compte toutes les valeurs ?). Ces problématiques sont abordées dans une série de séminaires organisés à l'Ircam [sem15].

On trouve également des travaux qui cherchent à faciliter la manipulation et la communication dynamique d'objets s'exécutant à des fréquences de calcul ou d'échantillonnage différentes avec des blocs audio de tailles variables [Ess12].

### 4.4.2 *Associer le temps symbolique au temps physique*

Les fonctions permettant de mettre en relation le temps en pulsation de la partition et le temps réel ont été étudiées dès les premiers langages pour l'informatique musicale. Il faut ajouter à cela le temps de la performance. Ce mapping permet de spécifier des changements de tempo dynamique, des techniques de phrasé tels que le rubato, des altérations rythmiques comme le swing. On trouve dans la littérature différents noms pour désigner ces opérations : *time maps* [Jaf85], *time deformations* [AK89], *time warps* [Dan97a]

Dannenberg propose dans le langage fonctionnel *Nyquist* le concept d'abstraction comportementale [Dan84b, Dan97b], qui permet d'effectuer des opérations s'adaptant automatiquement au contexte dans lequel elles sont réalisées. L'utilisateur peut définir sur un objet, un comportement par rapport à un type d'opération (par exemple la transposition). Ainsi l'utilisateur peut appliquer une opération sur plusieurs objets musicaux sans se préoccuper de l'implémentation réelle de l'opération sur chacun des objets. Cette idée d'abstraction comportementale fonctionne aussi pour les opérations temporelles appelées *time-warping*, qui projettent le temps en pulsation sur le temps réel. Les opérations de temporalité fines peuvent être codées sur les objets. Cela permet d'obtenir des résultats physiquement et musicalement cohérents une fois les objets étirés par les fonctions de *time-warping*.

Honing défend l'idée que dans une performance musicale la pulsation, le placement temporel expressif (comme le ralentissement dans les fins de phrases), la structure temporelle (la métrique) sont intrinsèquement liés, l'un ne pouvant pas être modélisé sans les autres [Hon01]. Il n'existe pas une relation simple entre positions des événements dans la partition et tempo de la performance. Honing propose une représentation souple pour la transformation du temps musical : les *timing function* (TIF) (cf. Figure 14).

L'expressivité temporelle d'une performance est vue comme la combinaison de fonctions qui correspondent à des variations de tempo, de position ou de phase au cours du temps. Ce temps peut être exprimé dans le repère absolu, dans le repère de la partition ou plus localement dans un repère qui est lui-même soumis à des variations temporelles.

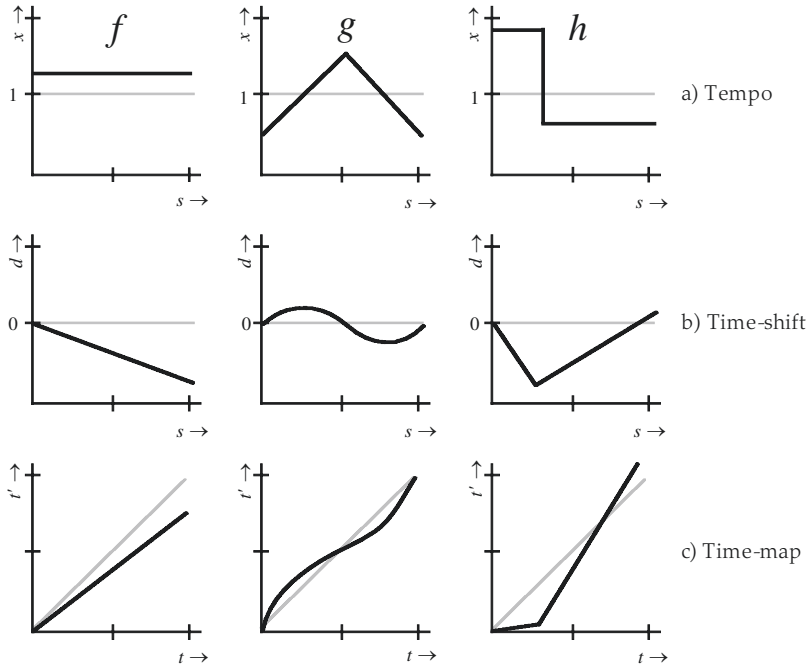



FIGURE 14: Schéma extrait de l'article de Honing [Hono1] qui représentent trois variations temporelles sous trois points de vue différents, (tempo, décalage temporel et position).

## 4.5 JOUER DES SCÉNARIOS DANS LE TEMPS

On s'intéresse dans cette section à la réalisation lors de la performance de structures temporelles de haut-niveau qu'on appellera scénario. Ce scénario peut être décrit à l'avance ou bien improvisé lors de la performance comme c'est le cas dans le live-coding.

### 4.5.1 *Organiser un scénario dans le temps*

Organiser le scénario d'une pièce est une étape essentielle du travail compositionnel. Il permet de définir les exécutions possibles de la performance. Il peut être composé de notes, de structures harmoniques, de fichiers audio, de programmes, etc. Il peut être linéaire, cyclique, ouvert, ou bien encore réactif. Nous présentons différents types de logiciels permettant de réaliser des scénarios.

Les séquenceurs classiques tels que *LogicPro*, *CuBase* ou *ProTools*, permettent d'organiser des pièces musicales et multimedia au cours d'un temps linéaire. *Ableton Live* est plus axé pour la performance en temps réel (cf. Figure 15). Il offre deux vues différentes, la vue *Arrangement* avec une notion de temps linéaire similaire à celle des logiciels précédents et la vue *Session* permettant d'organiser sous la forme de pistes différents matériaux (instruments, boucles audio ou symbo-

FIGURE 15: Vu session du logiciel Ableton Live.

liques, etc) qui peuvent être activés ou désactivés pendant l'exécution et qui se synchronisent sur une horloge globale. *Max4Live* ajoute davantage de possibilités d'interaction.

La *maquette* d'OpenMusic est un outil permettant d'organiser des objets musicaux au cours du temps mais elle n'offre aucune possibilité d'interaction [BGA06].

Le projet *Iscore* [DCA05] est un outil pour la composition assistée par ordinateur. Il permet au compositeur de construire et relier des modules de production sonore ou de contrôle avec des relations de logique temporelle inspirées des relations d'Allen [All83]. Cette approche est intéressante d'un point de vue compositionnel pour l'écriture de scénarios interactifs en particulier dans le domaine des spectacles vivants. L'interaction reste cependant limitée à des communications de paramètres et il n'y a pas de réelle considération dans le modèle sous-jacent d'un environnement musical dynamique dans le modèle. Par exemple, le tempo du musicien n'est pas une notion primitive dans *Iscore*.

Pour organiser une pièce interactive dans Max ou Pd, une solution commune mais peu souple au niveau temporel consiste à préparer une liste séquentielle de structures de données associées à des dates symboliques. Ces dates symboliques correspondent le plus souvent à un pivot de synchronisation dans la partition qui seront évaluées à la réception d'un message provenant d'un contrôleur ou d'un suiveur de partition. Puckette a introduit des structures de données plus complexes dans l'environnement de programmation de Pure-Data dans le but d'organiser plus facilement des informations dans le temps [Puc02]. On peut par exemple les associer à des objets graphiques où l'axe des abscisses représente le temps. L'écriture de scé-

narios temporels complexes reste néanmoins assez délicate à cause de la nature du langage *flot de donnée* peu adapté à l'expression de traitements hétérogènes au cours du temps.

### 4.5.2 *Le temps du live coding*

Le live coding est une pratique apparue récemment mêlant programmation et improvisation. Les performers écrivent des programmes et les exécutent pendant le temps de la performance. Nous présentons quelques langages adaptés à cette pratique mais dont les propriétés dynamiques sont également intéressantes dans un contexte plus large.

Les langages dynamiques tels que *SuperCollider* [McC96], *Chuck* [Wan08], Impromptu [Sor05] rencontrent aujourd'hui un succès certain auprès de la communauté en informatique musicale. Ce sont des langages textuels (par opposition à *Max* et *PureData*) adaptés à la programmation de processus algorithmiques et permettant de faire de la synthèse, du traitement audio et de la composition en temps réel.

*Chuck* offre à travers un langage flexible et intuitif des mécanismes pour la synthèse, l'analyse sonore et le contrôle de processus. Du point de vue de l'utilisateur, Chuck supprime la distinction habituellement faite entre le calcul audio et le contrôle. Chaque processus contrôle finement le déroulement temporel de son exécution grâce à l'opérateur `+=>` qui permet de « chucker » une durée (exprimée en échantillons, en seconde ou dans une unité définie par l'utilisateur, etc.) à la variable `now` correspondant à l'instant courant. On peut aussi se mettre en attente d'un événement particulier grâce à l'opérateur `=>`. Cette conception particulière du temps permet de faire cohabiter et de synchroniser des processus concurrents évoluant à des temporalités hétérogènes.

*SuperCollider* a été conçu afin de permettre la spécification des idées compositionnelles le plus directement possible dans un contexte de performance dynamique. *SuperCollider* est un langage similaire à *Smalltalk*, dynamiquement typé, orienté objet. À travers des structures de données adaptées, *SuperCollider* facilite la description d'un ensemble d'exécutions possibles plutôt qu'un comportement statique.

Les propriétés dynamiques et les performances temps réel de ces langages en font des outils adaptés au « Live Coding » (cf. Figure 16). Cette pratique aborde le temps musical d'une nouvelle manière puisque l'action de programmation est au coeur du processus compositionnel dans le temps de la performance. Autrement dit le temps de la conception du programme et le temps de son exécution sont confondus [SG10].

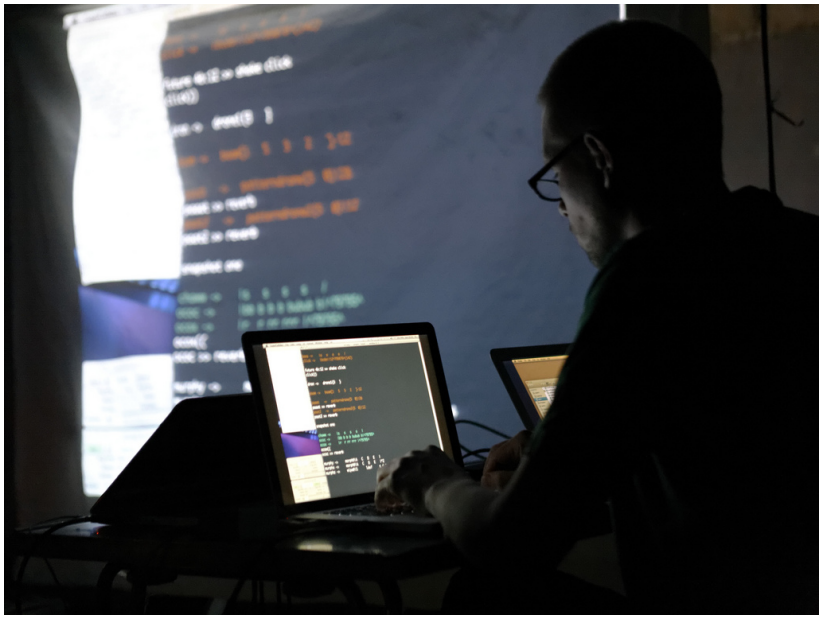


FIGURE 16: Performance de live coding (Thor Magnusson, 2012). L'écran est projeté pour que le public visualise la construction du programme responsable de la musique générée (photographie : Steve Welburn).

## 4.6 POSITIONNEMENT D'ANTESCOFO

Le but d'*Antescofo* est de faciliter la description de scénarios interactifs musicien-machine. Pour cela nous avons adopté un point de vue temporel original. À l'image des musiciens, *Antescofo* est capable pendant la performance de relier le temps symbolique de la partition au temps physique en tenant compte des variations inattendues de l'environnement. Cette capacité à gérer le temps de la performance est intégrée dans le processus compositionnel. Ces caractéristiques sont uniques à *Antescofo*.

*Antescofo* supporte la plupart des abstractions offertes par les langages dynamiques de l'informatique musicale. Il s'inspire des séquenceurs classiques mais avec un modèle de temps multiples et considère également les relations complexes entre pulsation et temps physique. Il faut ajouter à cela la prise en compte de l'indéterminisme de l'environnement musical dans lequel il s'exécute.

CALCUL AUDIO. Contrairement à la plupart des langages de programmation que nous avons cités dans ce chapitre, *Antescofo* ne permet pas d'effectuer de calculs audio. Cependant il entretient des liens étroits avec ce type de calculs :

– la machine d'écoute d'*Antescofo* analyse le signal sonore en entrée pour en déduire la position du musicien dans la partition,

- *Antescofo* est très souvent utilisé pour contrôler des processus sonores à l'extérieur,
- une des direction de recherche future est d'intégrer le calcul audio dans le langage.

FORMULA. Le langage d'*Antescofo* présente des similarités conceptuelles avec le système *Formula* mais il étend et généralise ses fonctionnalités, par exemple en permettant de préciser finement, pour une séquence donnée quelle stratégie de synchronisation adopter pour accompagner le musicien et comment gérer les erreurs. Ces stratégies s'adaptent ensuite dynamiquement à chaque nouvelle information issue du jeu du musicien pour obtenir un résultat musicalement cohérent.

TEMPS PERFORMATIF ET TEMPS RÉIFIÉ DE LA COMPOSITION. *Antescofo* s'inscrit dans le rapprochement entre le temps compositionnel et le temps performatif car il essaye d'allier au sein d'un même environnement de programmation des outils pour la composition (à travers des structures de données de haut-niveau, la possibilité de parler du temps de différentes manières) et des mécanismes pour la performance en temps réel (évaluation dynamique des expressions temporelles, adaptation temporelle).

MAX ET PURE DATA. Le langage impératif d'Antescofo, et la représentation explicite de la partition à suivre, permet de pallier aux problèmes causés par le style flot de données de Max. D'ailleurs, *Antescofo*, qui se compile sous la forme d'un objet Max ou PureData, est utilisé par les artistes pour pallier ce manque et contrôler d'autres objets Max.

ACCOMPAGNEMENT AUTOMATIQUE. Le logiciel *Antescofo* est d'abord apparu comme un outil de suivi de partition où l'utilisateur (le compositeur) choisit les actions à exécuter pendant la performance. C'est cette caractéristique qui le différencie des applications pour l'accompagnement automatique qui ne sont pas orientées vers la composition. Le développement du langage pour spécifier des programmes plus complexes et des interactions musicales plus fines, ajoute une nouvelle dimension au suivi de partition.

ASSOCIER LE TEMPS SYMBOLIQUE AU TEMPS DE LA PERFORMANCE. En combinant les différents éléments du langage d'*Antescofo*, événements et durées, l'utilisateur peut décrire des comportements temporels complexes d'une manière intuitive qui seront complètement déterminés dans le temps de la performance.

Le modèle utilisé pour l'estimation globale du tempo du musicien repose sur un modèle d'oscillateur auto-entretenu qui fournit une

très bonne estimation du futur. Ce modèle peut d'ailleurs être associé à n'importe quelle entrée du système (à travers la mise à jour d'une variable). Les actions séquencées du langage peuvent suivre le tempo global du musicien, le tempo d'une variable ou un tempo local exprimé à partir d'une expression que l'on peut manipuler explicitement comme l'objet d'un calcul et dont la variation peut être décrite au cours du temps, avec des structures de contrôle continu par exemple. De plus, les stratégies de synchronisation mises à disposition permettent un large choix de synchronisation avec les événements du musicien ou avec un autre processus, qui vont de la coordination synchrone à des coordinations souples et continues.

LIVE CODING. Le Live Coding et les applications visées par *Antescofo* nécessitent l'utilisation de constructions dynamiques qui, pendant la performance, s'adaptent respectivement aux instructions du programmeur ou aux variations de l'environnement.

Le modèle du temps musical dans la sémantique de ces langages dynamiques reste simple : il ne prend pas en compte la complexité temporelle de l'interprétation musicale. Cela rend donc plus difficile la programmation d'interactions fines et musicales entre le musicien et la machine.

Ces nouvelles pratiques ont cependant montré toute l'importance de pouvoir tester un code rapidement et d'interagir en temps réel avec un programme qui s'exécute. Pour *Antescofo*, ces capacités sont essentielles pour la composition, les répétitions et pour contrôler l'exécution du programme. Ainsi, il est possible d'évaluer au vol une expression, en parallèle avec l'exécution d'une partition augmentée, afin de changer l'état d'un calcul en cours. Il est également possible par exemple de pallier à une erreur de la machine d'écoute, en forçant la reconnaissance de l'événement attendu, ou de se repositionner à un endroit de la partition. Toutes ces commandes utilisées habituellement par le compositeur ou le réalisateur en informatique musicale (RIM) en répétition ou en concert, correspondent à des instructions internes du langage, qui ont donné lieu à des applications inattendues dans le domaine de l'improvisation par exemple. Elles augmentent considérablement l'utilisabilité du « système *Antescofo* ».

ORGANISER UN SCÉNARIO INTERACTIF La composition d'une œuvre mixte avec le langage d'*Antescofo* peut être vue comme la spécification d'un scénario interactif. Le système partage donc de nombreux points communs avec les séquenceurs qui permettent la construction de scénarios musicaux. Il se différencie cependant par ses caractéristiques dynamiques et interactives : les références temporelles peuvent être associées au temps élastique (événement et durée) du musicien. Le placement temporel, les délais entre les actions, la durée des processus de calcul et les tempos associés sont calculés par

des expressions arbitraires pouvant dépendre de l'environnement extérieur.

*Antescofo* met à disposition des compositeurs un langage leur permettant une réelle écriture de l'électronique, en permanence liée à celle destinée aux interprètes, et amène par cela un arsenal « temporel » leur permettant d'exprimer simplement des relations et des constructions hiérarchiques potentiellement très complexes. Il n'y a plus ici d'un côté un instrumentiste et de l'autre une machine, avec leur partition respective, mais un espace commun dans lequel peut se construire, dans la pluralité des temps mis en œuvre, un discours musical et des repères communs.

Deuxième partie

# LE LANGAGE ANTESCOFO

Cette deuxième partie présente le langage d'*Antescofo*. Le chapitre 5, décrit les principaux concepts du sytème et les éléments du langage permettant de spécifier des actions électroniques en interaction avec le jeu d'un musicien. Le chapitre 6 introduit le modèle temporel du système en particulier les différentes stratégies pour coordonner des processus électroniques avec un référentiel particulier. Une sémantique du langage est donnée dans le chapitre 7 et enfin le chapitre 8 offre un aperçu de l'implémentation du sytème *Antescofo*.

# 5 LANGAGE

Nous présenterons dans ce chapitre le langage d'*Antescofo*, qui permet de concevoir et de réaliser un scénario musical dans lequel les sons instrumentaux produits par un musicien humain, et électroniques sous le contrôle d'un programme, peuvent être en interaction permanente.

*Antescofo* est intégré dans le travail de nombreux compositeurs et réalisateurs en informatique musicale : depuis 2007, l'organisation temporelle de nombreuses œuvres, réalisées à l'IRCAM [1] mais aussi partout dans le monde, repose sur cet outil. Le logiciel est disponible sur le site du Forum Ircam [2].

Le chapitre est organisé de la façon suivante :

- Les sections 5.1 et 5.2 introduisent les concepts principaux du système et du langage.
- Nous détaillerons dans les sections 5.3 et 5.4 les expressions et les actions qui sont manipulables dans le langage.
- Dans la section 5.5 nous décrirons comment les actions peuvent être lancées et arrêtées.
- Enfin nous présenterons dans la section 5.6 les mécanismes de communication avec l'environnement.

## 5.1 INTRODUCTION

### 5.1.1 *Couplage entre un modèle probabiliste et un modèle réactif*

Le système *Antescofo* vise à retrouver la puissance symbolique de la notation musicale classique dans le cadre nouveau de la musique mixte, où il est nécessaire à la fois de définir les parties électroniques et d'exprimer les interactions complexes entre les musiciens et ces processus lors du concert. En 2008, Arshia Cont propose une architecture s'appuyant sur le couplage fort entre une machine d'écoute artificielle et un langage dédié temps réel dans [Cono8]. Ainsi, le compositeur écrit une *partition augmentée* comme un programme qui définit à la fois les événements du musicien qu'il faut reconnaître en temps réel et la spécification d'actions électroniques (cf. figure 17). Pendant le concert, le moteur d'exécution du langage évalue la partition augmentée et contrôle l'évolution temporelle des processus électroniques en fonction de l'environnement musical grâce à la machine d'écoute artificielle. Celle-ci est capable de suivre le jeu du musicien en com-

*partition augmentée = programme*

*performance = interprétation du musicien + réalisation de l'électronique + couplage*

1. Institut de Recherche et Coordination Acoustique/Musique, Paris.
2. http://forumnet.ircam.fr/

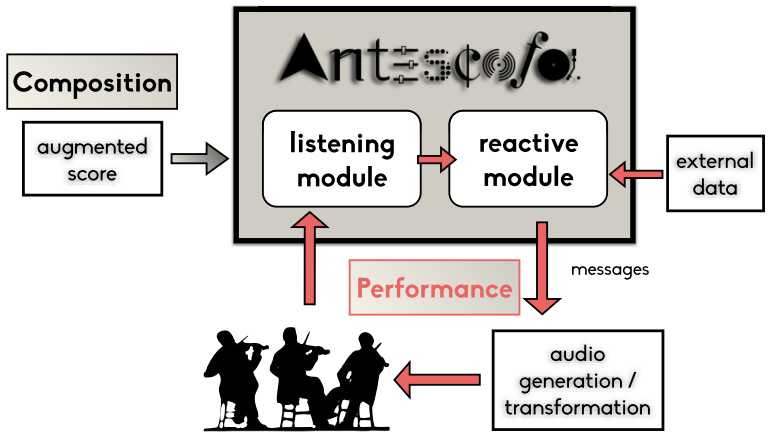



FIGURE 17: Architecture d'*Antescofo*. La partition sert à la fois à la machine d'écoute et à la machine réactive. Les événements reconnus par la machine d'écoute sont signalés à la machine réactive pour ordonnancer les actions

parant le spectre du flux audio avec un spectre généré à partir des informations de la partition. Le modèle utilisé par la machine d'écoute repose sur des semi-chaines de Markov cachées [Gué05].

### 5.1.2 *Programmer les interactions musicales*

Bien que le suivi de partition soit un domaine très étudié en informatique musicale, les aspects compositionnels et réactifs associés à une machine d'écoute sont restés peu explorés jusqu'à présent. La dimension performative de l'accompagnement automatique qui doit être introduite dans la partition augmentée donne lieu à de nouveaux défis dans la communauté des systèmes temps réel :

- la gestion de relations temporelles multiples (succession, simultanéité, et durée) et d'horloges multiples (tempo, temps physique) ;
- la spécification de « synchronisations musicales » subtiles entre le musicien et la machine ;
- la garantie d'un comportement musical de la machine en interaction avec un environnement non déterministe.

Nous nous attacherons à mettre en perspective ces défis autour de notre solution, le langage dynamique d'*Antescofo*. Ces trois points ont amené à développer des mécanismes nouveaux qui font l'originalité de ce système, validés lors de la création de nouvelles pièces ou la réécriture de pièces préexistantes.

## 5.2 PRÉSENTATION SUCCINCTE DU LANGAGE

Nous présentons dans cette section les notions constitutives du langage, afin d'en avoir une vue générale. Nous reviendrons plus en

détail sur chacune de ces notions, dans les sections et le chapitre suivant.

*partition augmentée*

Une partition *Antescofo* correspond à la spécification de la partie instrumentale et des actions électroniques à réaliser lors de la prestation musicale. Cette spécification est donnée en entrée de la machine d'écoute et du module réactif (cf. figure 17). Pendant le concert, le module réactif réagit aux données de la machine d'écoute (position courante du musicien dans la partition + estimation du tempo) et de l'environnement extérieur pour exécuter des actions d'accompagnement. Un des principaux atouts d'*Antescofo* réside dans le fait qu'un compositeur peut exprimer la temporalité des événements et des actions relativement :

- aux événements du musiciens,
- au tempo du musicien ou à un tempo calculé,
- à un mixte des deux précédents.

Au sein d'une partition, le compositeur peut par exemple décider de déclencher des actions au moment de la détection d'un événement musical, après un certain délai évalué à la volée, exprimé en pulsations ou en secondes. Il peut également grouper des actions de façon hiérarchique, les lancer en parallèle, les itérer et définir des comportements temporels.

### 5.2.1 *Événements instrumentaux à reconnaître*

*le vocabulaire des événements musicaux reconnus*

La partie instrumentale est décrite à l'aide de mots clés correspondants à des événements musicaux : notes, accords, trilles, glissandi, etc. [Cono8].

Dans l'exemple suivant deux événements sont spécifiés :

```
NOTE C4 1 e1
CHORD (C4 F#4) 1/2 e2
```

Ils correspondent à la partition du musicien et devront être reconnus par le système lors sa prestation. Le premier est un do `C4`, qui dure un temps, le deuxième est un accord de do et fa# `(C4 F#4)` qui dure `1/2` temps. Des labels optionnels, `e1` et `e2` permettent de se référer à ces événements ailleurs dans la partition.

### 5.2.2 *Actions*

La syntaxe du langage permet de définir 2 catégories d'actions. On peut leur associer un label dont la portée est toujours globale. Une *action atomique* correspond à une instruction élémentaire qui s'exécute en temps 0 (hypothèse synchrone) et une *action composée* structure dans le temps un ensemble d'actions. Les actions composées peuvent s'imbriquer les unes dans les autres. La durée d'une action composée correspond à l'intervalle de temps qui sépare son déclenchement de l'exécution de la dernière action atomique dans la hiérarchie.

### 5.2.3 *Délais*

Chaque action peut être précédée d'une expression optionnelle qui correspond au *délai* séparant son exécution de la détection de l'évènement précédent ou l'exécution de l'action précédente.

La séquence suivante :

```
NOTE C4 2
  1/2 action1
  3   action2
NOTE D5 1
```

spécifie que l'*action1* sera exécutée $1/2$ temps après la reconnaissance de la note `C4` et l'*action2* sera exécutée 3 temps après l'exécution de l'*action1*.

*plusieurs interprétations d'un délai*

Il y a plusieurs façons de compter ces délais pendant l'exécution. On peut considérer par exemple que l'*action2* s'exécute $(1/2+3)$ temps après la détection du premier événement, ou alors, si ce délai est supérieur à la durée de ce premier événement, on peut exécuter l'action $((1/2+3)-2)$ temps après la détection du deuxième événement `NOTE D5` ; cela dépend de la stratégie de synchronisation choisie par le compositeur (cf. section 6.4.4). Lorsqu'aucun délai n'est spécifié, ou lorsque le délai est évalué à 0, l'action qui suit est exécutée immédiatement après l'action ou l'événement précédent dans le même instant logique, cf. section 5.2.5.

### 5.2.4 *Tempo et pulsation*

*plusieurs unités de temps différentes pour exprimer un délai*

Dans le langage d'*Antescofo* un délai est une expression et peut être spécifié en pulsations (par rapport à un tempo) ou en temps physique (en secondes ou millisecondes). Le tempo, exprimé en nombre de pulsations par minute, spécifie « la vitesse d'écoulement du temps de la partition ». Nous reviendrons plus en détail sur la modélisation de temps dans *Antescofo* à travers la notion de tempo dans le chapitre 6.

Un tempo peut être associé à une action composée, il permet alors d'interpréter les délais exprimés en pulsation. Par défaut, le tempo réfère au tempo estimé en temps réel à partir des événements du musicien reconnus par la machine d'écoute.

### 5.2.5 *Instants logiques*

Un instant logique correspond à un laps de temps considéré comme nul, pendant lequel des actions s'exécutent sans interruption. Lors de l'exécution, toute action du système est associée à un instant logique auquel on a attribué une date. Cette notion a été introduite dans le moteur d'exécution pour structurer l'exécution du programme *Antescofo* dans le temps, assurer l'abstraction synchrone et réduire les approximations temporelles.

*causes d'un instant logique*

Trois événements peuvent déclencher un nouvel *instant logique* :
- la reconnaissance d'un événement musical par la machine d'écoute,
- l'affectation d'une variable depuis l'environnement extérieur,
- l'expiration d'un délai.

Dans un même instant logique, les actions synchrones sont exécutées séquentiellement en respectant l'ordre d'écriture de la partition. Dans *Antescofo*, la mise en parallèle n'est donc pas commutative. Cette « non-commutativité » est une propriété similaire à celle que l'on retrouve dans les langages de programmations synchrones [Hal98] et assure le déterminisme des calculs.

### 5.2.6 *Coroutines*

*coroutine = exécution d'une action composée*

Nous appelons coroutine d'*Antescofo*, une unité de code correspondant à l'exécution d'une action composée. L'exécution d'une coroutine peut-être suspendue ou relancée en certains points prédéfinis (entre les délais). Cette notion est similaire à celle introduite par Conway [Con63]. Pendant l'exécution, elles sont instanciées à chaque lancement d'une action composée. Leur rôle consiste à gérer tous les paramètres dynamiques de l'action associée.

La table 1 présente les principales notions du langage et leur utilité musicale. Ces notions sont détaillées dans les sections suivantes.

## 5.3 EXPRESSIONS

Soit les expressions apparaissent comme paramètres des actions et sont évaluées au moment du lancement de l'action, soit elles apparaissent comme le délai précédent une action. Dans ce cas, l'expression est évaluée après l'exécution de l'action précédente ou après la reconnaissance de l'événement précédent.

En tant que paramètre d'une action, les expressions peuvent être utilisées comme nom du receveur d'un message, comme argument d'un message, comme période d'une boucle, comme condition logique d'une conditionnelle, comme argument d'un appel de fonction ou de processus, ou comme spécification de tempo ou d'une stratégie de synchronisation, etc.

### 5.3.1 *Valeur*

Le langage est dynamiquement typé et gère les types suivants : entier, flottant, chaîne de caractères, symbole, tableau, dictionnaire, fonction, processus, coroutines. Plus de 160 fonctions prédéfinies sont disponibles et l'utilisateur peut définir les siennes.

*fonction d'ordre supérieur*

Les fonctions sont des valeurs de première classe. Il faut noter que les fonctions, tant prédéfinies que définies par l'utilisateur, sont

TABLE 1: Abstractions dans le langage

| *Abstractions* | *Description et usage* |
|---|---|
| événement | décrit un événement joué par le musicien reconnaissable par la machine d'écoute |
| action atomique | calcul ou message de durée nulle |
| action composée : | contrôle l'ordre et la temporalité des exécutions |
| - group | sequence des actions dans le temps (phrase musicale) |
| - loop | itération une séquence d'actions |
| - curve | interpolation de paramètres continus dans le temps (geste musical) |
| processus | permet d'abstraire des séquences et de les rejouer dynamiquement à la demande |
| label | permet de désigner un événement musical ou une action |
| nom de variable | permet l'accès à une donnée |
| nom de fonction | dénote une valeur fonctionnelle |
| structure de donnée : | permet d'organiser les données sous forme de |
| - tab | tableau |
| - map | dictionnaire |
| - NIM | représentation symbolique d'une fonction interpolée |
| liaison lexical des variables | fournit l'accès à la valeur dans le contexte de définition |
| liaisons dynamique des attributs temporels | héritage des attributs temporels du contexte d'appel |
| attribut de synchronisation | stratégie d'alignement temporel d'une séquence d'actions avec la performance du musicien |
| attribut de rattrapage d'erreur | gestion des événements manqués ou non-reconnus |
| délai | durée temporelle permettant de spécifier la date d'une action relativement à l'action ou l'événement qui la déclenche |
| tempo | spécification de l'unité de temps utilisée pour mesurer les délais |

des valeurs de première classe. On peut donc définir des fonctions d'ordre supérieur. Les fonctions sont implicitement curryfiées et l'application partielle d'une fonction retourne une fonction.

*processus valeur de première classe*

Les processus sont aussi des valeurs de première classe, au même titre que les fonctions. Ils constituent l'analogue des fonctions dans le domaines des actions. On peut écrire des processus d'ordre supérieur et des processus récursifs. Mais les processus ne sont pas curryfiés.

*coroutine*

Les coroutines correspondent à des actions en cours d'exécution. Ce sont des valeurs de première classe et permettent de terminer une action particulière et plus généralement d'interagir avec l'exécution associée. On peut ainsi récupérer la valeur courante d'une variable locale d'un processus qui se déroule en parallèle, via la valeur qui la réfère. Plusieurs coroutines peuvent correspondre à la même action. Cela correspond à des exécutions parallèles de la même action, ce qui peut arriver si on lance plusieurs fois le même processus par exemple.

*valeurs scalaires et composées*

Les valeurs se partagent en deux classes : les valeurs simples, encore qualifiée de scalaires, et les valeurs composées qui comprennent les tableaux, les dictionnaires et les NIM.

Les valeurs composées sont mutables : il est possible de modifier les éléments d'une valeur composée. Elles sont allouées dynamiquement et implicitement par le système d'exécution. Un compteur de référence permet de désalouer ces structures implicitement, dès qu'elles ne sont plus utilisées.

### 5.3.2 *Variables utilisateurs*

Comme dans les langages impératifs, les variables *Antescofo* correspondent à des zones mémoires dont le contenu peut changer au cours de l'exécution. Elles sont par défaut globales, accessibles en lecture et écriture n'importe où dans la partition. On peut cependant spécifier qu'une variable est locale à une action composée, limitant ainsi sa portée et sa durée de vie. La liaison des variables est statique (lexicale) mais les paramètres de synchronisation et de tempo sont liés dynamiquement. Un identificateur de variable débute par le caractère `$`.

Les affectations sont considérées comme des actions atomiques ordonnées d'un point de vue logique comme dans la plupart des langages de programmation mais également positionnées dans le temps et donc liées à un instant. De cette façon, on peut construire pour chaque variable un historique de ses valeurs avec les dates associées. Grâce à cet historique, le compositeur peut alors accéder à la fois aux valeurs passées et aux dates d'affectations. Ainsi, `$v` correspond à la dernière valeur du « flot de données » et

```
[date]:$v
```

correspond à la valeur de `$v` à la date `date`. Cette date peut s'exprimer de trois manières différentes :

*trois références à un instant passé*

- par leur rang de mises à jour : l'expression `[n#]:$v` retourne la n-ième valeur passée de `$v`, avec $n \in \mathbb{N}$ (`[0#]:$v = $v`);
- par une durée en secondes : l'expression `[f s]:$v` retourne la valeur de `$v` f secondes avant l'évaluation de cette expression, avec $f \in \mathbb{R}^+$ ;
- par une durée en pulsations : l'expression `[p]:$v` retourne la valeur de `$v` p pulsations avant l'évaluation de cette expression, avec $p \in \mathbb{R}^+$ ;

Le programmeur peut spécifier la taille de l'historique de chaque variable. Le système renvoie une valeur indéfinie lorsqu'on essaye d'accéder à une valeur qui est en dehors des limites de l'historique.

Les variables peuvent être mises à jour explicitement dans la partition par l'affectation ou via un mécanisme permettant de mettre à jour les variables depuis l'environnement extérieur. Ainsi *Antescofo* peut réagir non seulement aux événements musicaux notifiés par la machine d'écoute mais aussi à toutes sortes d'événements en provenance de l'environnement.

### 5.3.3 *Accès aux dates d'affectations*

Deux expressions permettent aux compositeurs d'accéder à la date d'un instant logique associé à l'affectation d'une variable `$v`. L'expression `@date([n#]:$v)` retourne la date en seconde de la n-ième dernière affectation de `$v` tandis que `@rdate([n#]:$v)` retourne la date en pulsations. Par exemple, `@date([0#]:$v)` correspond à la date de la dernière affectation.

### 5.3.4 *Variables du système*

`$RT_TEMPO`, `$LOCAL_TEMPO`, `$BEATPOS`, *etc.*

Un certain nombre de variables sont gérées par le système et peuvent être accessible en lecture dans la partition. On accède au tempo global du musicien via la variable `$RT_TEMPO`. Le tempo local d'une coroutine est disponible à travers la variable `$LOCAL_TEMPO`. `$PITCH`, `$BEATPOS` et `$DUR` correspondent respectivement à la hauteur (ou à la liste des hauteurs pour un accord), la position dans la partition et la durée du dernier événement détecté.

Il existe un ensemble de variables qui correspond à la position courante dans un référentiel temporel particulier. Nous détaillerons leur définition et leur comportement dans le chapitre 6.

## 5.4 ACTIONS

Nous détaillerons dans cette section les différents types d'actions qui peuvent être spécifiés dans le langage ainsi que leurs propriétés.

### 5.4.1 *Actions atomiques*

Une action atomique est toujours exécutée à l'intérieur d'un seul instant logique et correspond :

- soit à une affectation de variable (effective immédiatement) :

```
$v := 2+3
```

- soit à une primitive interne, par exemple la commande qui permet de terminer une action (le mot clé suivi du label de l'action) :

```
abort group1
```

- soit à un message envoyé à l'environnement extérieur, par exemple un message contrôlant des paramètres d'un module de synthèse sonore (identifiant du receveur suivi des paramètres) :

```
synth $v 2 @sin($x)
```

Le receveur d'un message peut être calculé dynamiquement.

```
$x := 3
@command("synth" + $x ) 20
```

Ce programme va envoyer 20 au receveur `synth3`.

D'autres actions atomiques existent : envoi d'un message OSC, écriture dans un fichier, appel d'un processus, évaluation d'une assertion et toutes les commandes qui permettent de piloter le système (activation de la machine d'écoute, changement de paramètre interne, positionnement dans la partition, etc.). Nous ne détaillerons pas plus ces actions atomiques : le lecteur intéressé peut se référer au manuel de référence du langage [GCE15].

### 5.4.2 *Actions composées*

Les *actions composées* sont des structures de contrôle permettant de séquencer d'autres actions (atomiques ou composées) dans le temps ; elles sont simples, répétées ou continues.

Ces actions permettent au compositeur de construire facilement des hiérarchies imbriquées et organisées en fonction de leur comportement.

Des *attributs* permettent d'associer à une action composée : un nom, un tempo local, une stratégie de synchronisation et de rattrapage d'erreurs. S'ils ne sont pas spécifiquement définis, les attributs sont hérités de l'action composée englobante.

Les actions composées peuvent s'exécuter sur plusieurs instants logiques ; entre deux instants elles sont en attente d'un événement extérieur ou de l'expiration d'un délai dans une file de l'ordonnanceur (cf. section 8.5).

*terminaison d'une action composée*

Une action composée peut se terminer :

- naturellement avec le déclenchement de la dernière de ses actions atomiques,

– lorsque une commande `abort` lui est destinée (cf. section 5.5),
– prématurément lorsqu'une condition de terminaison est vérifiée.

Les conditions de terminaison sont des attributs optionnels spécifiés après les mot-clés `until`, `while`, ou `during`. Les commandes `until` et `while` permettent de spécifier une condition logique qui sera testée à chaque activation de la coroutine correspondant à l'exécution de l'action composée. Si cette condition est vraie pour `until` ou fausse pour `while`, l'exécution se termine immédiatement. La clause `during` permet de spécifier une durée de vie maximale pour l'exécution de cette action. Cette durée est exprimée en temps logique, relatif ou physique.

**Groupe.** La construction `group` séquence au sein d'un même bloc des actions partageant des propriétés communes de tempo, de synchronisation, de gestion des erreurs, etc. Elle facilite l'écriture de phrases indépendantes pour former une polyphonie, sans devoir calculer l'entrelacement entre les actions.

```
group { actions* }
```

**Boucle.** La construction `loop` est similaire à `group` mais la séquence d'actions est itérée après un certain délai (`period` dans l'exemple suivant). Le délai, évalué à chaque itération, correspond à la période de la boucle quand il est constant. Un phénomène de tuilage peut se produire si la durée du bloc d'actions est supérieure à la période.

Pour une boucle, les clauses de terminaison sont évaluées à chaque début d'itération. La période d'une itération $n$ est évaluée au début de l'itération $(n-1)$.

```
loop period { actions* } until expression
```

**Interpolation.** La construction `curve` effectue des calculs d'interpolation de flottants. L'utilisateur peut choisir le type et le pas (`@grain`) d'interpolation. Il peut spécifier les valeurs de références (les paliers), la durée entre chaque palier, les variables affectées et l'action à exécuter à chaque pas d'interpolation. Le pas et la durée entre les paliers peuvent être spécifiés en temps relatif ou en temps absolu. Cette construction permet de contrôler l'évolution de paramètres de processus musicaux de façon « quasi continue ».

```
curve grain action
[var*]
{           [expression*] interpolation type
  ( delay [expression*] interpolation type )*
    delay [expression*]
}
```

Il existe plus d'une trentaine de type d'interpolation, qui sont illustrés à la figure 18.

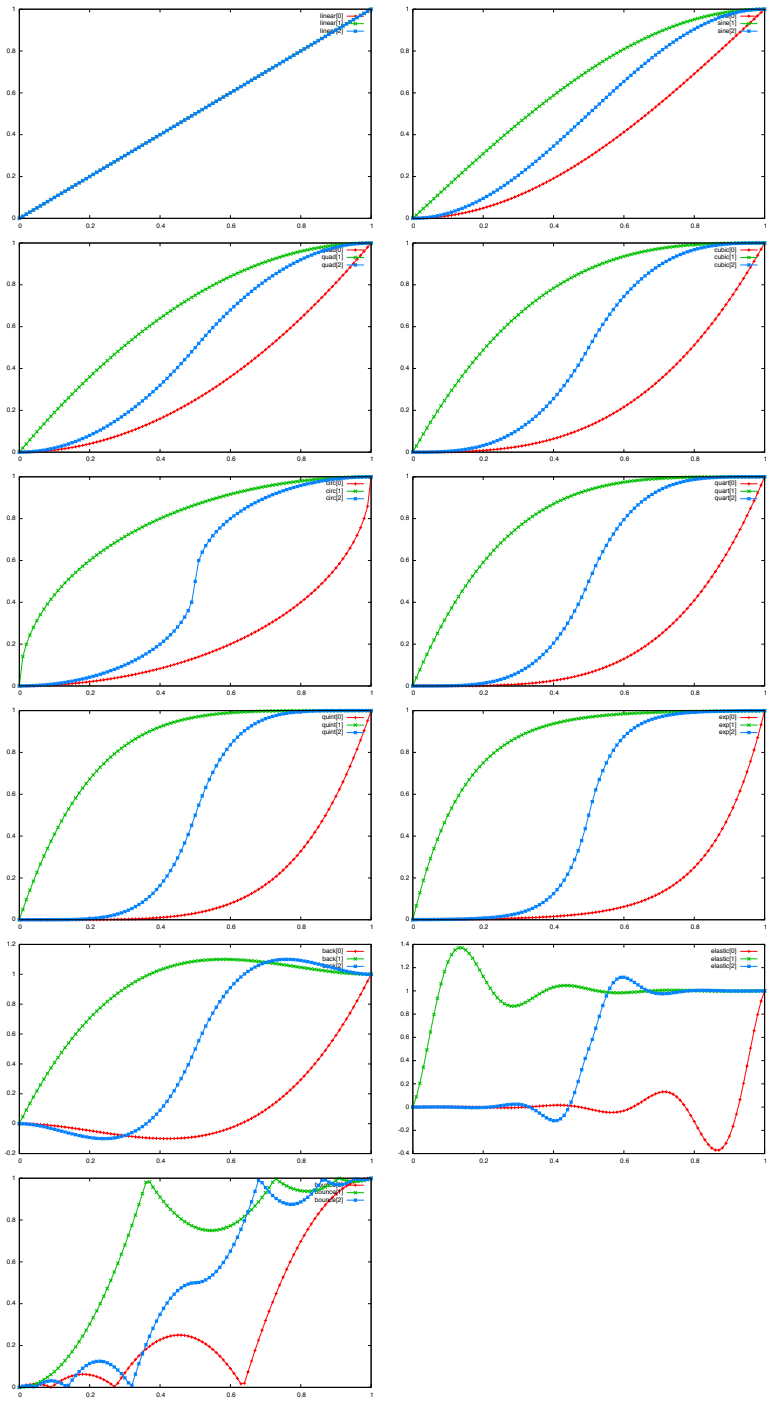



FIGURE 18: Illustrations des différents types d'interpolation disponibles pour les curves et pour les NIM. Les types les plus simples correspondent à des polynômes. Les types les plus complexes impliquent des fonctions trigonométriques et exponentielles.

### 5.4.3 *Les processus*

Les processus sont des actions créées dynamiquement. Un processus est spécifié par la construction `@proc_def`. Tous les noms de processus sont introduits par `::`. Après sa définition, un processus `::P` peut être appelé avec des arguments. Cet appel exécute alors une instance de ce processus sous la forme d'un groupe d'actions (plusieurs instances d'un même processus peuvent s'exécuter en parallèle).

```
@proc_def ::echo($pitch, $d)
{
   $d/3 loop $d/3
   {
      play $pitch
   } during [2#]
}
NOTE C4 2
  ::echo("C4", 2)
NOTE D3 1
  ::echo("D3", 1)
```

Dans l'exemple précédent le processus `::echo` répète deux fois la note qui l'a déclenché, réparties pendant la durée de celle-ci. On utilise pour cela une boucle, lancée après un délai de $d/3, de période $d/3 qui envoie le message `play $pitch` à chaque tour de boucle, au nombre de deux (`during[2#]`).

Nous listons ici quelques propriétés spécifiques aux processus :

- un processus peut aussi être appelé dans une expression. L'instance du processus est alors lancée et la valeur retournée correspond à la coroutine réalisant le corps du processus ;
- les processus peuvent être récursifs ;
- leurs arguments sont calculés au moment du lancement du processus ;
- il est possible de terminer tous les processus d'un nom donné ou d'une instance spécifique :

  ```
  $p1 := ::P()
  $p2 := ::P()
  $p3 := ::P()
  10.0 abort $p1
  1 abort ::P
  ```

  Ici `$px` représente une coroutine associée à l'exécution de P correspondante.
- les processus sont des valeurs de premier ordre : on peut passer un processus `::P` en argument d'un autre processus. Par exemple :

  ```
  @proc_def ::Tic($x) {
    $x play TIC
  }
  @proc_def ::Toc($x) {
    $x play TOC
  ```

```
}
@proc_def ::Clock($p, $q) {
  :: $p(1)
  :: $q(2)
  3 ::Clock($q, $p)
}
```

L'appel `Clock(::Tic, ::Toc)` jouera TIC une pulsation après l'appel, TOC deux pulsations après l'appel, et un nouvel appel sera lancé 3 pulsations après le premier en inversant `TIC` et `TOC`. Au-delà de cet exemple illustratif, cette paramétrisation des processus permet la construction de schémas compositionnels complexes.

### 5.4.4 *Conditionnelle*

La conditionnelle permet d'exécuter un bloc ou un autre en fonction de la valeur d'une expression.

```
if (expression) {
        actions*
}else{
        actions*
}
```

Différentes variantes de la conditionnelle existent comme l'instruction `switch...case`.

### 5.4.5 *Forall*

Alors que la construction `loop` répète une séquence d'actions dans le temps en fonction d'une période spécifiée, l'action `forall` instancie dans le même instant logique un groupe pour chaque élément d'un ensemble donné. Par exemple dans le code suivant,

```
$t := [1, 2, 3]
forall $x in $t
{
  action $x
}
```

la construction `forall` permet de lancer en parallèle une action pour chaque élément du vecteur `$t`. La variable `$x` prend la valeur de l'élément correspondant dans le vecteur.

La forme générale de la construction `forall` est

```
forall $variable in expression
{
    actions*
}
```

où l'évaluation de *expression* est associée à un vecteur ou à un processus (cf. section 5.4.3). Dans ce dernier cas, on itère sur toutes les instances du processus, la variable prenant à chaque fois la valeur

d'une de ces instances. La structure `forall` accepte aussi les dictionnaires. L'itération se fait alors avec deux variables.

```
$m := MAP { (key1, val1), (key2, val2), (key3, vval3) }
forall $k, $v in $m
{
    actions*
}
```

### 5.4.6 *La structure de contrôle whenever*

La structure de contrôle `whenever` permet de lancer un groupe d'actions lorsqu'une certaine condition logique est vérifiée.

```
whenever (predicate)
{
    action*
} until (expression)
```

La condition `predicate` est un prédicat arbitraire. Elle est réévaluée à chaque fois qu'une variable de ce prédicat est mise à jour, si la condition n'a pas déjà été évaluée dans le même instant logique (cf. section 8.5.1).

La structure de contrôle `whenever` échappe à la nature séquentielle de la partition traditionnelle. En effet les actions qui la composent ne sont pas associées statiquement à un événement du musicien mais peuvent dynamiquement être associées à un ou plusieurs événements dépendants d'un calcul interne et/ou de données reçues de l'environnement extérieur. Cette structure de contrôle permet donc de simplifier l'écriture de certains mécanismes réactifs et rend possible l'interaction avec l'environnement extérieur sans passer par la machine d'écoute.

Attention, il ne faut pas confondre le `whenever` avec les constructions de sous-échantillonnage que l'on trouve dans les langages synchrones (comme le when en *Lustre*). Dans *Antescofo* l'exécution du corps du `whenever` correspond au lancement d'une coroutine et plusieurs instances de ce corps peuvent être en cours d'exécution à un moment donné.

## 5.5 DÉMARRER ET ARRÊTER DES ACTIONS

L'exécution d'une action est déclenchée (éventuellement après l'expiration d'un délai) :

- par l'occurrence d'un événement ;
- par le déclenchement d'une action ;
- par l'activation d'un whenever consécutivement à une affectation de variable ;
- par une itération de `loop` ou de `forall` ;
- par un appel de processus.

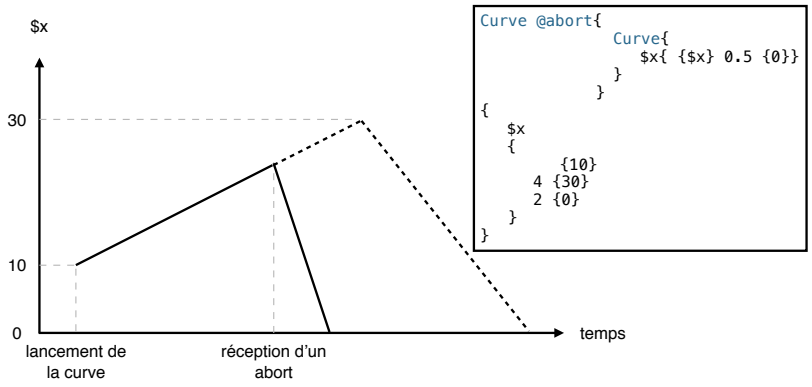



FIGURE 19: Exécution possible d'une `curve`. Un message `abort` signale la terminaison de la curve avant que celle-ci n'ait pu finir son exécution normale. Le *handler* est alors exécuté.

Comme il a été indiqué plus haut, il est possible de terminer une action composée avant d'avoir exécuté toutes les actions qui la composent, en utilisant des clauses de terminaisons.

Il est aussi possible de terminer une action composée par une commande `abort`. Cette commande prend en argument soit un label, auquel cas toutes les actions en cours d'exécution portant ce label ainsi que ces actions filles sont terminées, soit une valeur qui réfère à une coroutine particulière et dans ce cas, seule la coroutine et ses actions filles sont terminées. Associée à l'attribut `norec`, la commande `abort` entraîne uniquement la terminaison l'action concernée et non les actions filles.

La création dynamique via la commande `whenever` et la terminaison via la commande `abort` sont des mécanismes très utiles pour réagir dans un environnement non-déterministe. Il est ainsi possible de lancer une série d'actions en réaction à un événement et de stopper ces actions à l'occurrence d'un autre événement.

La gestion de la terminaison pose cependant un problème : cette terminaison peut survenir n'importe quand pendant l'exécution et peut donc laisser le système dans un état incohérent. Pour faire face à ce problème, *Antescofo* introduit la notion de *handler* : un groupe d'actions exécuté à la terminaison d'une action composée, par la commande `abort`. Un exemple classique est de ramener un paramètre échantillonnant une `curve` à une valeur prédéterminée (cf. Figure 19).

### 5.5.1 *Les motifs temporels*

La structure `whenever` permet de réagir à un événement. Dans son utilisation classique, le compositeur spécifie à l'aide d'une combinaison de variables une condition qui correspond à l'événement déclencheur lorsqu'elle est vérifiée. Les motifs temporels facilitent la spécifi-

cation d'événements complexes comme par exemple une succession d'événements avec des contraintes temporelles particulières. Les événements peuvent être ponctuels (`EVENT`) mais peuvent aussi avoir une durée (`STATE`). L'exemple ci-dessous montre comment définir l'évènement complexe `pattern::P` correspondant à la configuration suivante : la variable `$x` est supérieure à un seuil égal à 10 pendant 0.5s puis la variable `$y` prend la valeur 1 avant qu'une pulsation ne soit passée. On utilise ensuite ce pattern dans la spécification d'une condition pour associer une réaction à cet événement.

```
@pattern_def pattern::P
{
  STATE $x where($x>10) during 0.5s
  before 1
  EVENT $y where($y=1)
}
...
whenever (pattern::P) { action* }
```

Les motifs temporels que l'on peut spécifier et leur sémantique sont présentés dans [GE+14].

Les motifs ressemblent à des expressions régulières, interprétées dans le temps plutôt que sur une séquence. Ils sont restreints aux motifs qui peuvent être reconnus causalement mais étendus sur un alphabet infini dont les éléments correspondent à des événements élémentaires ou à des intervalles. Le tout dépend de conditions abstraites portant sur des paramètres des éléments de l'alphabet (date d'un événements, durée d'un interval , etc.).

On peut par exemple définir de manière concise un événement complexe `E` qui fait référence à sa propre occurrence tel que « la répétition de la même note dans un intervalle de 3/2 pulsations tel qu'il n'y a pas d'occurrence de `E` dans les 5 pulsations précédentes ».

Les motifs temporels d'*Antescofo* se compilent au vol en une imbrication de `whenever` réalisant le calcul de la condition de déclenchement au cours du temps. La définition du motif temporel complexe ne nécessite donc pas d'extension du langage et nous ne la présenterons pas plus avant. Le lecteur intéressé peut se référer à l'article [GE+14] et au manuel de référence d'*Antescofo* [GCE15].

## 5.6 COMMUNICATIONS AVEC L'ENVIRONNEMENT EXTÉRIEUR

Plusieurs mécanismes de communication entre l'environnement extérieur et *Antescofo* ont été mis en place pour permettre la réalisation des différentes tâches d'interaction.

### 5.6.1 *Communication avec Max et PureData*

*Antescofo* est utilisé en général comme un objet de l'environnement de programmation Max ou PureData. Les communications de Max ou Pd vers *Antescofo* (mis à part le flux audio) se réalisent grâce à des commandes spéciales tandis que celles d'*Antescofo* vers l'environnement hôte sont réalisées à travers les messages.

MESSAGES Les actions qui correspondent aux messages dans le langage d'*Antescofo* respectent les conventions des messages Max et Pure Data. Lorsque l'action `synth 20` est exécutée dans *Antescofo*, tous les receveurs *synth* de l'environnement hôte, reçoivent alors la valeur 20.

En respectant ces conventions, *Antescofo* facilite le travail d'intégration des utilisateurs dans ces environnements.

LES COMMANDES Les commandes correspondent à des messages qui sont envoyés depuis Max ou Pd directement à l'objet *Antescofo*. Ils permettent notamment de charger une partition, de lancer la machine d'écoute, de simuler la partition, de gérer la position courante du suivi, etc. Elles sont utilisées aussi bien pendant les phases de composition et de répétition que pendant le concert.

Il existe différents types de commandes qui permettent de se replacer dans la partition à une position particulière, donnée via un label ou une position en pulsation. L'utilisateur peut choisir d'exécuter toutes les actions avec ou sans envois de messages depuis la position courante jusqu'à la position demandée en respectant l'ordre idéal d'exécution de la partition.

La commande *setvar* est utilisée pour changer la valeur d'une ou plusieurs variables dans l'environnement d'*Antescofo* et a les mêmes conséquences qu'une affection dans le langage notamment pour les structures `whenever` et les motifs temporels. La commande *playstring* permet d'exécuter un programme *Antescofo* dans l'environnement courant sans avoir à recharger de nouvelle partition.

Toutes les commandes sont « internalisées » comme des actions atomiques dans le langage.

### 5.6.2 *Communication* Open Sound Control

*Open Sound Control* (OSC) est un protocole de communication largement utilisé dans la communauté des arts interactifs. L'envoi et la réception de messages OSC peuvent être gérés directement depuis une partition *Antescofo*. Une fois les ports et les noms de receveurs paramétrés, ils s'utilisent comme les autres messages.

# 6 LA GESTION DU TEMPS DANS ANTESCOFO

Dans le chapitre précédent nous avons vu les éléments du langage permettant de construire des séquences d'actions temporelles qui sont déclenchées avec la survenue d'un événement. Ces actions ont une date implicite qui indique quand l'action doit s'exécuter, de manière similaire aux notes d'une partition classique. Ces dates sont virtuelles et leur réalisation dépendra d'une interprétation lors de la performance.

Ce chapitre présente cette problématique et détaille les nombreux mécanismes du langage permettant d'interpréter une date lors d'une performance afin de jouer avec le musicien et plus généralement dans n'importe quel référentiel temporel :

- Les sections 6.1 et 6.2 introduisent quelques notions générales sur l'interprétation et le tempo.
- La section 6.3 présente les différentes estimations de la position du musicien maintenue par le système lors de la performance.
- Les stratégies de synchronisation sont décrites dans la section 6.4.
- La section 6.5 précise comment créer un référentiel à partir de la mise à jour d'une variable.
- Enfin la section 6.6 introduit la notion de tempo explicite définie par l'utilisateur.

## 6.1 INTERPRÉTATION MUSICALE ET MUSIQUE MIXTE

En lisant une partition, un musicien est capable de comprendre l'organisation d'objets musicaux tels que les notes, les accords, les phrases et autres structures temporelles décrites par le compositeur. Certaines caractéristiques de ces objets telles que les dates de réalisation, les durées, les stratégies de synchronisation ou les nuances ne sont pas totalement déterminées ; cette indétermination laisse le champ libre à l'interprétation.

> *Ces valeurs sont virtuelles et ne deviennent réelles qu'au moment où l'interprète les produit. On ne peut rigoureusement parler d'interprétation que lorsqu'il y a une incertitude sur la valeur réelle qui va intervenir.*
>
> *Philippe Manoury* [Man12]

L'interprétation musicale est donc fortement dynamique et l'évaluation des durées dépend de chaque prestation et des stratégies du musicien, conditions difficilement formalisables. Cependant, lorsque plusieurs musiciens jouent ensemble, le résultat sonore reste cohérent

en dépit du non-déterminisme de la partition. Cette cohérence est rendue possible grâce à des propriétés d'écoute active, d'anticipation et de stratégies de synchronisations adaptées.

## 6.2 TEMPO

Pour interpréter ces durées *Antescofo* s'appuie sur deux référentiels distincts que sont le temps physique $\mathcal{P}$ et le temps relatif $\mathcal{T}$. Le référentiel associé au temps relatif est défini à partir d'un tempo. Le tempo, exprimé par exemple en nombre de pulsations par minute (BPM), spécifie le « passage du temps du musicien » dans le référentiel associé au temps physique [MZ94]. Pendant la prestation du musicien, la machine d'écoute d'*Antescofo* estime à chaque nouvel événement son tempo. Elle utilise pour cela un modèle cognitif simulant le comportement du cerveau humain [LJ99]. Ce tempo général est utilisé pour calculer les délais qui dépendent de ce référentiel. Dans la phase d'écriture de la partition cette donnée est accessible à travers la variable `$RT_TEMPO`.

Dans *Antescofo* les compositeurs peuvent introduire leurs propres référentiels en spécifiant un tempo $T_{\mathcal{T}}$ sous la forme d'un attribut pour toute action composée (groupe, boucle, interpolation). Ce tempo est une expression arbitraire réévaluée chaque fois que nécessaire pour ajuster la manière dont on « compte » le temps dynamiquement. Les actions filles héritent du tempo de l'action englobante à moins qu'un nouveau tempo ne leur soit attribué.

*tempo et position*

Idéalement, il y existe une relation exacte entre le tempo, la date $d$ d'un événement ou d'une action, et sa position dans une partition augmentée : l'intégrale de l'inverse du tempo entre le début de la partition et la date $d$, donne sa position dans la partition.

$$\text{position} = \int_0^d \text{tempo}\, dt \tag{1}$$

Cette relation idéale n'est vérifiée que pour les événements qui dépendent d'un tempo assuré par une machine ou calculé en temps différé. Or *Antescofo* étant utilisé dans un contexte temps réel, le tempo extrait de la machine d'écoute est une mesure globale du flux passé du musicien et la relation n'est qu'approximative. Par exemple, les notes peuvent localement être jouées un peu en avance ou un peu en retard par rapport au tempo estimé. Une des forces d'*Antescofo* est de permettre de *gérer le hiatus entre les deux manières de compter le temps* :

*gérer le temps événementiel et le temps continu*

- par événement,
- par écoulement,

suivant l'approche qui sera musicalement la plus adaptée selon le compositeur.

## 6.3 POSITIONS COURANTES

*3 progressions d'horloges*

La variable `$NOW` correspond à la date de l'instant courant en secondes. Les variables `$T_NOW` et `$A_NOW` sont deux différentes estimations de l'instant courant du musicien dans la partition en pulsations. La variable `$L_NOW` est le temps en pulsation uniquement selon l'estimation du tempo[1]. Elles sont définies de la manière suivante. Au début de la partition :

$$\$NOW_0 = \$L\_NOW_0 = \$T\_NOW_0 = \$A\_NOW_0 = 0$$

À un instant quelconque de l'exécution, soit $e_n$ le dernier événement reconnu par la machine d'écoute (aux dates $\$NOW = \$NOW_{e_n}$ et $\$L\_NOW = \$L\_NOW_{e_n}$), et $e_{n+1}$ l'événement suivant, $p_n$ et $p_{n+1}$ leurs positions respectives dans la partition (en nombre de pulsations), avec $delta$ le temps en pulsation depuis la reconnaissance de $e_n$ défini selon l'équation suivante :

$$delta = (\$NOW - \$NOW_{e_n}) * \$RT\_TEMPO/60$$

alors :

$$\begin{aligned} \$L\_NOW &= \$L\_NOW_n + delta \\ \$T\_NOW &= min(p_n + delta, p_{n+1}) \\ \$A\_NOW &= p_n + delta \end{aligned}$$

Les valeurs de `$T_NOW` et `$A_NOW` diffèrent donc lorsque l'estimation d'arrivée du prochain événement est dépassée ; `$T_NOW` stagne à cette valeur tant qu'un événement n'a pas été détecté, alors que la variable `$A_NOW` continue d'avancer en suivant le tempo sans tenir compte de la non-survenue de l'événement. L'estimation du temps du musicien selon `$A_NOW` est donc plus fiable quand un événement a été manqué (le musicien ne l'a pas joué ou bien la machine ne l'a pas identifié) mais elle doit parfois « revenir en arrière » lorsque sa prédiction est en avance contrairement à `$T_NOW` (cf. figure 26).

Tout nouvel instant logique commence par la mise à jour des variables `$T_NOW`, `$A_NOW`, `$L_NOW` et `$NOW`. Elles sont utilisées en interne pour le calcul des délais, des tempos dynamiques, des stratégies de synchronisation et pour la gestion des historiques des variables.

## 6.4 STRATÉGIES DE SYNCHRONISATION ET GESTION DES ERREURS

Lorsqu'un compositeur écrit de la musique, il intègre le fait que les musiciens seront libres d'interpréter certains paramètres musicaux de son œuvre. Le temps est une propriété qui n'est pas totalement

1. Les lettres T A et L des noms des variables `$T_NOW`, `$A_NOW` et `$L_NOW` font respectivement référence aux stratégies de synchronisation `@tight`, `@ante`, `@loose` présentées section 6.4.1.

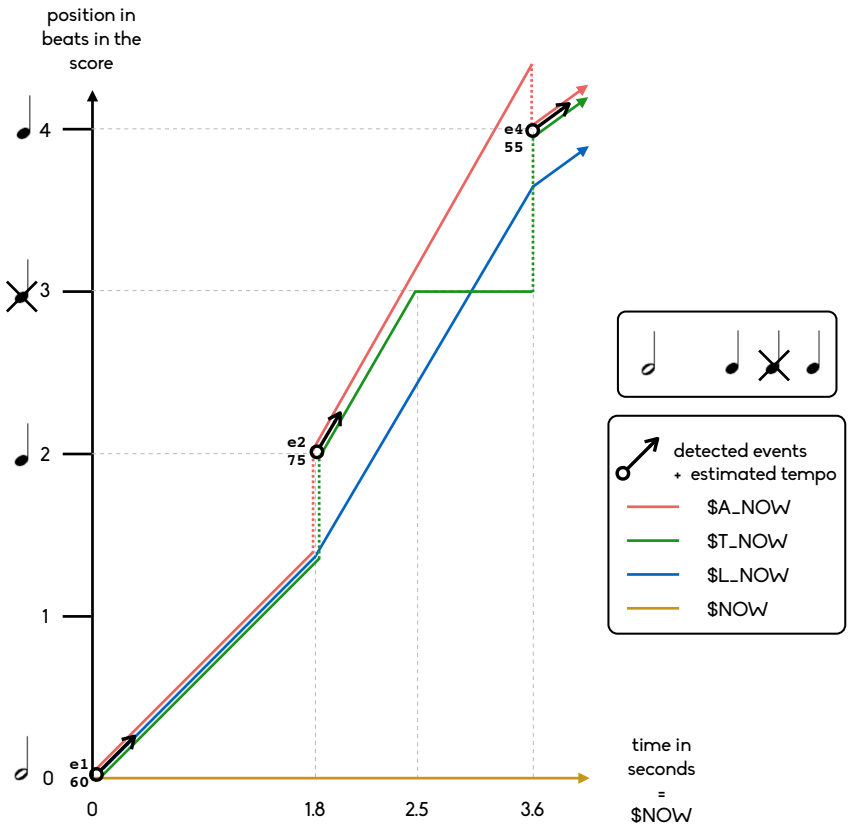



FIGURE 20: Comportement des variables temporelles `$T_NOW`, `$A_NOW`, `$L_NOW` et `$NOW` dans le cas d'une interprétation simple où le troisième événement est manquant. Le temps en secondes est représenté en abscisses, la position en pulsations en ordonnées. Les vecteurs permettent de visualiser la date à laquelle l'événement correspondant a été détecté, la pente du vecteur représentant le tempo estimé à ce moment là

déterminée lors de la phase compositionnnelle, et qui se trouve en partie soumise à l'interprétation du musicien.

Dans un contexte de musique mixte, pour synchroniser les actions électroniques au jeu d'un musicien il faut à la fois tenir compte :

- du contexte musical (prédéfini),
- du tempo réel du musicien,
- de l'occurrence des événements musicaux,
- des erreurs éventuelles du musicien et de la machine d'écoute.

La sémantique du langage d'*Antescofo* permet de modéliser et d'expliciter l'ensemble de ces caractéristiques à travers la mise à disposition de stratégies de synchronisation et de rattrapage d'erreurs. Le compositeur peut ainsi définir un comportement pour les actions électroniques qui imite les stratégies d'un musicien à l'écoute.

### 6.4.1 *Stratégies de synchronisation avec tempo non-adaptatif*

Les musiciens combinent différentes techniques pour se synchroniser : un tempo partagé, une écoute active et des informations visuelles (gestes, regards). On retrouve dans *Antescofo* une modélisation des deux premières techniques mais pas de la troisième. En contrepartie, le système possède un temps de réaction bien meilleur que celui du musicien et peut être considéré comme nul d'un point de vue perceptif. Cependant il ne suffit pas seulement de réagir rapidement pour bien se synchroniser, il faut aussi anticiper les événements et la bonne manière d'anticiper dépend du contexte musical. Les problèmes de synchronisation et de coordination musicales ont beaucoup été étudiés dans la littérature [LJ11] [WEBV14]. L'estimation du tempo dans *Antescofo* est d'ailleurs le fruit de l'une de ces études [Laro1].

Le langage d'*Antescofo* permet d'expliciter précisément comment une séquence d'actions se synchronise en temps réel avec les événements du musicien. Nous sommes convaincus que pour bien coordonner les actions électroniques au jeu d'un musicien, il ne suffit pas d'implanter « *le* » bon modèle de tempo ou « *la* » bonne stratégie de synchronisation. La stratégie de synchronisation dépend du contexte musical et c'est donc au compositeur de décider comment les phrases électroniques seront associées aux événements détectés.

Dans *Antescofo*, lorsqu'on écrit une séquence d'actions dans un bloc (`group`, `loop` ou `curve`), on précise les délais qui séparent deux actions successives. Si l'on note ces délais en pulsations relativement au tempo global alors on peut appliquer à cette séquence l'une des quatre stratégies de synchronisation suivantes :

#### *Stratégie lisse (loose)*

Par défaut, la stratégie de synchronisation d'un bloc est la stratégie `@loose`. De manière analogue à la variable `$L_NOW`, la vitesse d'écoule-

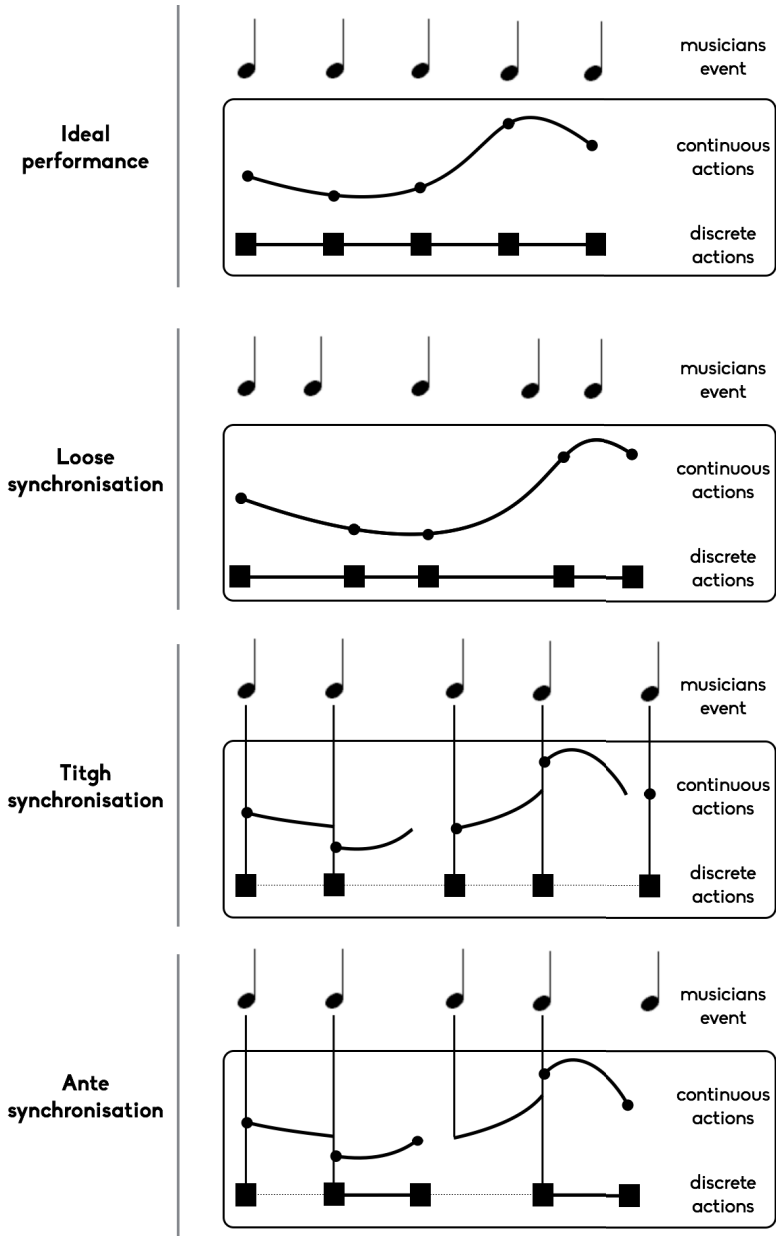



FIGURE 21: Comportement des actions continues (courbes) et discrètes (carrées) en fonction de la stratégie de synchronisation choisie. Le premier cas correspond à une interprétation idéale (le musicien joue exactement au tempo de la partition) ; le choix de la stratégie n'a aucune influence. Les trois autres cas correspondent aux trois stratégies pour une même interprétation non idéale : la deuxième et la quatrième notes sont jouées en avance, la troisième et la cinquième en retard. La stratégie `loose` transforme continument une action continue, mais l'alignement des actions et des événements est perdu. La stratégie `tight` préserve cet alignement, mais introduit des discontinuités dans les actions continues. La stratégie `ante` aligne les actions avec les événements quand ceux-ci ne sont pas en retard.

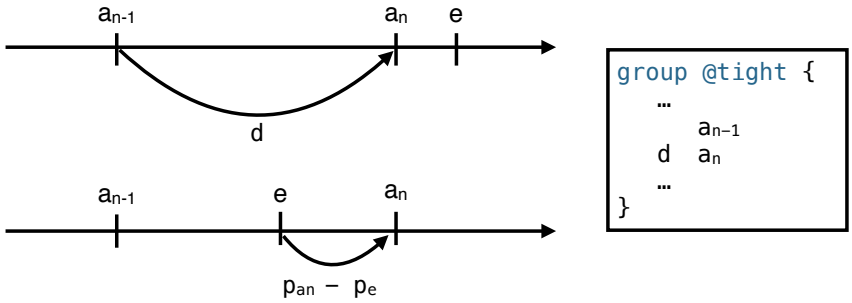



FIGURE 22: Comportements possibles pour l'ordonnancement de l'action $a_n$ qui doit être exécutée d pulsations après $a_{n-1}$. Si l'événement $e$ qui suit dans la partition doit arriver avant l'écoulement du délai alors on attend le temps nécessaire $(p_{a_n} - p_e)$ à partir de cet événement.

ment des délais entre les actions du bloc ne dépend que de l'estimation du tempo en temps réel, autrement dit, la position des actions n'est pas mise à jour à la détection d'un nouvel événement, seul le tempo change. La position des actions peut donc se décaler pendant la performance par rapport à celles des événements telles que décrites dans la partition. Ce type de synchronisation est lisse et sans discontinuité pour les actions continues (cf. figure 21).

*Stratégie événementielle (tight)*

La stratégie `@tight` d'un bloc d'actions permet de maintenir la position des actions au plus près de celles des événements, chaque action étant exécutée après l'action ou l'événement qui la précède.

Soit $a_n$ une action à la position $p_{a_n}$ d'un bloc de stratégie `@tight`, spécifiée avec un délai d après l'action précédente $a_{n-1}$ de position $p_{a_{n-1}}$. Soit $e$ le dernier événement dont la position $p_e$ est telle que $p_e \leqslant p_{a_n}$. Si $p_{a_{n-1}} < p_e \leqslant p_{a_n}$ alors $a_n$ est exécutée $(p_{a_n} - p_e)$ pulsations après la détection de $e$, sinon si $p_e \leqslant p_{a_{n-1}} < p_{a_n}$ alors $a_n$ est exécutée d pulsations après l'exécution de $a_{n-1}$. La recherche de l'événement synchronisant est effectuée dynamiquement après l'exécution de $a_{n-1}$ et l'évaluation de l'expression de d (cf. Figure 22)

Les constructions `curve` avec une stratégie `@tight` pourront effectuer des sauts discontinus pour rattraper un retard ou s'arrêter à une valeur en attendant l'événement synchronisant (cf. figure 21). L'avancement de la position courante dans un bloc avec une stratégie `@tight` est analogue à l'avancement de la variable temporelle `$T_NOW`.

*Stratégie anticipante (ante)*

La stratégie `@tight` peut avoir des effets indésirables de ralentissement dans une situation d'interaction forte entre le musicien et l'élec-

tronique. En effet, avec cette stratégie, bien que la machine réagisse quasi instantanément aux événements du musicien, celui-ci ne pourra jamais réagir instantanément aux événements de la machine car elle sera toujours en attente du musicien. Il aura donc tendance à ralentir à cause du manque d'initiative de la part de la machine. Pour remédier à ce phénomène une stratégie anticipative `@ante` est proposée.

Si une action $a_i$ a la même position théorique dans la partition qu'une note $n_i$ du musicien, alors $a_i$ sera exécutée à l'occurrence du premier événement parmi les deux suivants :

- l'expiration du délai entamé depuis la détection de la note précédente $n_{i-1}$,
- la détection de la note $n_i$.

Le comportement est analogue à l'avancement de la variable temporelle `$A_NOW`. Cette stratégie imite l'attitude d'un autre musicien qui anticipe les éléments synchronisés et installe une véritable interaction entre le musicien et la machine. L'effet de cette stratégie sur les actions continues est très similaire à la stratégie tight (cf. figure 21).

### 6.4.2 *Stratégies anticipatives d'adaptation du tempo*

Les stratégies de synchronisation précédentes ne proposent pas de véritable interaction entre le musicien et les actions électroniques. Soit la stratégie ne dépend que du tempo du musicien entraînant parfois un décalage de positions que seul le musicien peut rattraper, soit elle est complètement réactive aux événements créant des discontinuités dans la temporalité du processus. Dans tous les cas, le tempo du bloc utilisant ces stratégies est toujours égal au tempo estimé du musicien.

Cela ne suffit pas pour répondre à plusieurs problèmes rencontrés lors de plusieurs créations musicales. Les échanges avec les utilisateurs du système ont montré la nécessité de créer un réel dialogue dans la coordination des processus électroniques avec le jeu du musicien en combinant anticipation des événements et adaption continue de la position.

Les stratégies présentées dans cette section visent à répondre à ce problème. Elles sont qualifiées de stratégies anticipatives et visent à calculer dynamiquement et localement le tempo d'une séquence pour une synchronisation lisse et sans à-coups, suivant des indications précises du compositeur sous la forme d'attributs de blocs.

#### *Cibles statiques (static target)*

*événemetns pivots*

Certains événements dans une partition sont des *événements pivots* pour la coordination entre musiciens. On pense en particulier au début et fin de phrase mais pas seulement. Les musiciens sont capables pendant la performance d'adapter leur vitesse pour être parfaitement synchrones sur ces événements pivots.

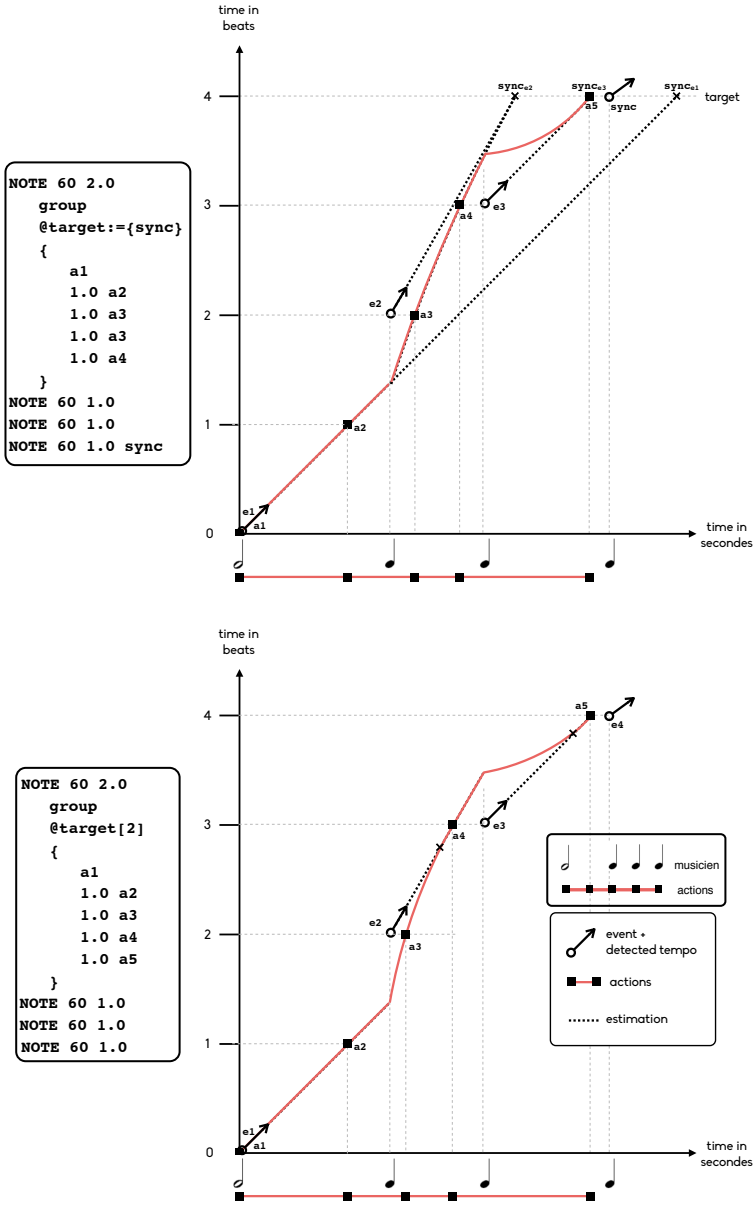



FIGURE 23: Exemple d'exécution de partitions avec des stratégies de synchronisation avec cibles. Les graphiques montrent la relation entre le temps relatif des actions et des événements (en ordonnée) par rapport au temps absolu (en abscisse). Les actions sont représentées par des carrés et le trait continu symbolise la progression des actions. Les événements sont symbolisés par des vecteurs où la pente correspond à l'estimation du tempo. La pente (le tempo) des actions est calculée à chaque réception d'un nouvel événement. Elle correspond à une fonction quadratique qui dépend de la différence entre la position de l'électronique et celle du musicien et de la contrainte de synchronisation (synchronisation en e4 pour la stratégie adaptative statique et fenêtre de synchronisation de 2 pour la stratégie dynamique).

À travers la stratégie de synchronisation proposée ici, *Antescofo* peut imiter ce comportement dans le séquencement des phrases électroniques. Le compositeur peut associer pour chaque bloc d'actions, une liste d'événements pivots. Pendant la performance, le tempo local du bloc est dynamiquement ajusté pour respecter ces contraintes temporelles (cf. Figure 23). Lorsque la séquence d'actions est lancée, sa positon locale `pos` est initialisée à la valeur de la variable `$A_NOW` et son tempo local est initialisé à `$RT_TEMPO`. Ensuite, à chaque fois que le système réactif d'*Antescofo* reçoit une nouvelle valeur de la position ou du tempo du musicien, le tempo local au bloc est re-calculé pour que la position locale du bloc suive une trajectoire parabolique (dans un repère où le temps est porté en abscisse, la position en ordonnées) qui vérifie les trois points suivants :

- la trajectoire passe par le point courant (position dans le bloc à l'instant courant) ;
- la trajectoire passe par l'événement synchronisant (le pivot) ;
- la tangente (tempo) en ce dernier point est égale au tempo du musicien.

Dans l'exemple suivant :

```
NOTE 60 2.0
   group @target:={e5, e10}
   {
      actions...
   }
   events...
NOTE 45 1.2 e5
   events...
NOTE 55 1.2 e10
```

le tempo local du groupe est d'abord calculé en fonction de l'estimation de la date d'arrivée de $e5$ puis de $e10$. Cette estimation varie au cours du temps au fur et à mesure des informations fournies par la machine d'écoute. Quand l'estimation varie, le tempo local est recalculé.

Cette stratégie de calcul du tempo est orthogonale à l'utilisation des stratégies `loose`, `tight` et `ante`, pour la synchronisation aux positions correspondant aux événements synchronisants. Par exemple si une telle stratégie anticipative est combinée avec une stratégie tight alors une synchronisation de type *rendez-vous* sera réalisée à chaque point de synchronisation (discontinuité si les actions sont en retard, attente de l'événement si les actions sont en avance).

La synchronisation des actions et des événements aux positions correspondant aux pivots est définie en fonction des stratégies `@loose`, `@tight`, `@ante`. Par exemple, en choisissant de combiner stratégie anticipative avec cibles statiques et stratégie `@tight`, l'exécution du bloc, s'il est en avance, sera gelée jusqu'à l'arrivée de l'événement pivot. S'il est en retard un saut de position sera effectué pour rattraper le musicien.

*Cibles dynamiques (Dynamic Target)*

Dans le cas des cibles dynamiques on ne spécifie plus une liste d'événements synchronisants mais un horizon temporel relatif de synchronisation exprimé en pulsations ou en secondes. Comme dans la stratégie précédente, lorsque la séquence d'actions est lancée, sa position locale `pos` est initialisée à la valeur de la variable `$A_NOW`, son tempo local est initialisé à `$RT_TEMPO`. À chaque fois qu'une nouvelle valeur de la position ou du tempo du musicien est disponible, alors le tempo local est re-calculé pour que la position locale du bloc suive une fonction parabolique. La fonction permet une convergence en tempo et en position mais la cible est mouvante et dépend cette fois-ci du paramètre (la fenêtre) donné par l'utilisateur. Ce paramètre correspond au temps nécessaire pour converger si la différence entre la position du musicien et la position locale du bloc est égale à 1.

```
NOTE 60 2.0 e1
   group @target := [2s]
   {
      actions...
   }
      events...
```

Plus la valeur du paramètre sera grande plus le temps de synchronisation sera long. Ainsi, l'utilisateur peut choisir précisément la vitesse de synchronisation ; la valeur 0 étant équivalente à une synchronisation `@tight`. De plus, il est possible de varier ce paramètre pendant l'exécution de la séquence car il peut apparaître sous la forme d'une expression qui est évaluée à chaque calcul. Ces calculs seront détaillés dans la section 7.8.1.

### 6.4.3 *Conservatif vs progressif*

Les stratégies de synchronisation qui influent sur le déroulement temporel des structures group, loop et curve peuvent se coordonner par rapport à une estimation de la position du musicien. On peut choisir avec les attributs `@conservative` ou `@progressive` si cette estimation correspond à la variable `$T_NOW` ou `$A_NOW`.

On préfèrera utiliser l'option conservative dans les cas où les variations de tempo sont faibles et/ou dans des situations où la qualité du suivi est localement imprécise même s'il reste globalement correct. En effet cette option a le mérite d'atténuer les erreurs de la machine d'écoute lorsque le déroulé temporel est relativement prévisible. Par contre dans les situations de rubato exagéré, l'option `@progressive` sera privilégiée. L'accompagnement ralentit ou s'arrêtera à partir du moment où l'estimation de la date de l'arrivée de l'événement suivant est dépassée.

La stratégie `@ante` est équivalente à une stratégie `@tight` combinée avec l'attribut `@progressive`.

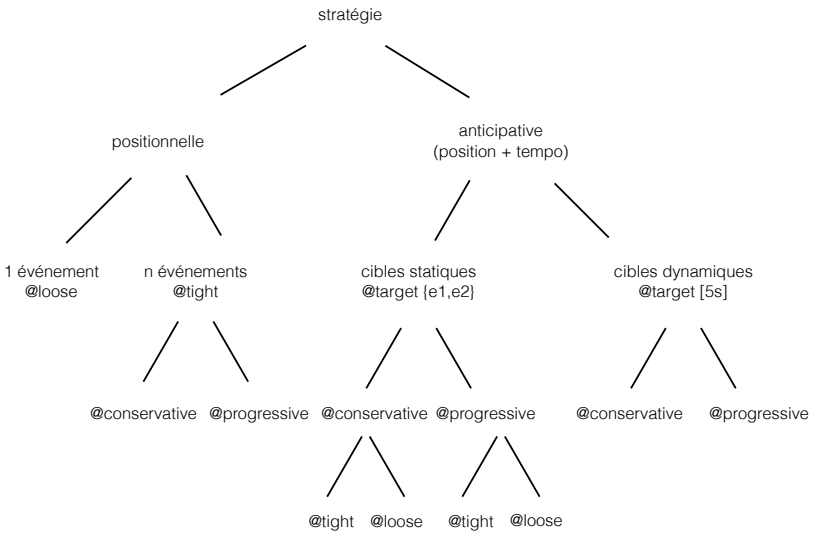



FIGURE 24: Panorama des stratégies permettant de synchroniser un bloc d'actions avec le tempo et les événements du musicien.

La figure 24 permet de visualiser toutes les combinaisons possibles entre les stratégies de synchronisation existantes. On distingue d'abord les stratégies de synchronisation dont le tempo est toujours égale à celui du musicien, des stratégies où le tempo s'adapte aux variations temporelles du musicien. Les premières se différencient ensuite par le nombre d'événements sur lequel elles se raccrochent (1 pour `@loose`, n pour `@tight`). Les stratégies avec cibles ou avec fenêtre composent les stratégies adaptatives. Les attributs `@conservative` ou `@progressive` peuvent s'appliquer sur toutes les stratégies qui prennent en compte la position des événements du musicien une fois lancée. Enfin, les stratégies avec cibles statiques se combinent avec les stratégies non-adaptatives pour décider du comportement aux positions qui correspondent aux cibles.

### 6.4.4 *Stratégies de rattrapage d'erreurs*

Dans un contexte de concert, de manière analogue aux systèmes cyber-physiques, il faut gérer les cas où l'environnement ne suit plus le scénario attendu. Différents cas d'erreur peuvent être rencontrés : la machine d'écoute peut ne pas détecter un événement, confondre un événement avec un autre ou passer trop vite à l'événement suivant. Il faut également considérer les éventuelles erreurs du musicien. Dans tous les cas, on souhaite non seulement que le système continue à fonctionner, mais aussi qu'il réagisse le plus musicalement possible. La machine d'écoute est robuste à beaucoup d'erreurs du musicien, elle peut par exemple suivre correctement la partition malgré certaines fausses notes.

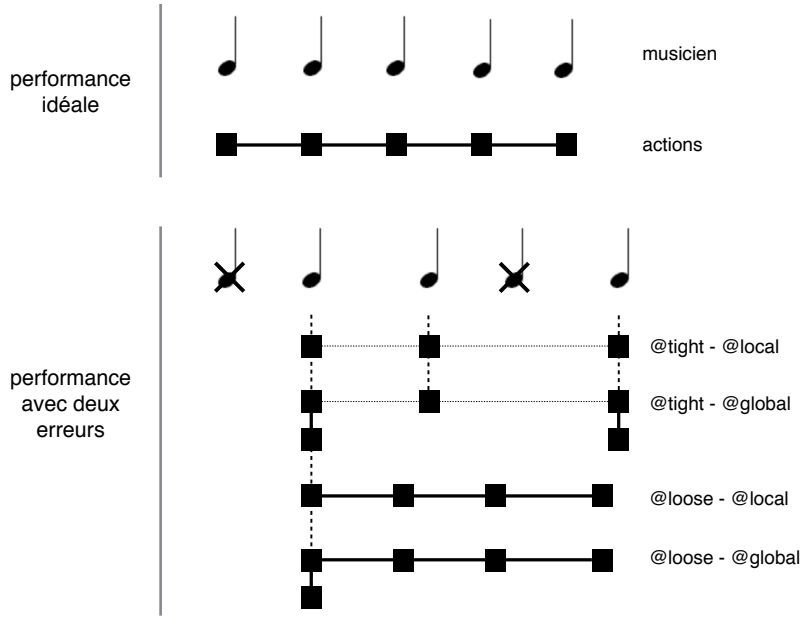



FIGURE 25: Comportements possibles d'un bloc d'actions en fonction des combinaisons des stratégies de synchronisation `@loose` et `@tight` et des stratégies pour la gestion des événements manqués `local` et `@global`.

La machine réactive ne reçoit de la machine d'écoute que des notifications des événements reconnus et ne détecte donc une erreur que lorsqu'on lui signale la reconnaissance d'un événement positionné après l'événement attendu dans la partition.

C'est au compositeur de définir à travers des attributs `@local` et `@global` (par défaut) le comportement des actions qui dépendent d'un événement raté. Une action *globale* sera lancée même en retard, si l'événement associé n'est pas détecté, c'est-à-dire à la détection de l'événement suivant. Une action *locale* ne sera jamais lancée si l'événement auquel elle correspond n'est pas détecté (cf. Figure 25).

Un bloc d'actions avec une stratégie `@loose` n'est directement concerné par la stratégie d'erreur que si l'événement qui le lance est manqué. On parcourt les actions (en les exécutant si l'attribut est `@global`) du bloc jusqu'à ce que la position de l'action courante soit supérieure à l'événement détecté. Pour les blocs `@tight`, on réitére cette procédure à chaque événement manqué.

Deux exemples simples permettent de comprendre l'intérêt de ces stratégies de rattrapage d'erreur : Une action peut correspondre à un changement nécessaire de l'environnement, comme par exemple éteindre les lumières au début d'une représentation. Si ce changement ne peut se faire à temps, il faut néanmoins l'exécuter et l'action doit être globale.

Une action peut activer d'un effet uniquement destiné à la note qui déclenche cette action. Dans ce cas si la note est ratée, l'action n'a pas lieu d'être. Elle doit donc être spécifiée comme locale.

## 6.5 CRÉER UN RÉFÉRENTIEL TEMPOREL À PARTIR DE LA MISE À JOUR D'UNE VARIABLE

Dans le moteur d'exécution, on déduit plusieurs référentiels temporels à partir des événements du musicien reconnus par la machine d'écoute. Ces référentiels sont utilisés pour le calcul des durées qui dépendent également des stratégies de synchronisation.

Nous avons généralisé la notion de référentiel temporel du musicien en permettant à l'utilisateur de créer et manipuler plusieurs référentiels en parallèle. C'est à travers les mises à jour d'une variable que l'utilisateur pourra créer un nouveau référentiel.

À chaque nouvelle mise à jour de la variable, une estimation du tempo (on utilise par défaut le même algorithme que pour celui du musicien) et un calcul de la position courante seront effectués. Ces mises à jour peuvent être réalisées depuis l'environnement extérieur ou dans la partition. Ensuite, il suffit d'associer la variable à la structure de contrôle temporelle (group, loop, etc.) pour que le déroulement temporel se coordonne par rapport à ce référentiel en spécifiant la stratégie de synchronisation choisie.

Il faut spécifier un tempo initial et une période à laquelle cette variable `$v` va être mise à jour. On peut récupérer la valeur du tempo (`$v.tempo`), de la position de la dernière mise à jour (`$v.position`), de la période (`$v.periode`) et des différentes estimations de la position courante de cette variable (`$v.anow` et `$v.tnow`). On peut modifier les valeurs de tempo, de position et de fréquence.

Par exemple, dans la partition suivante, on initialise les deux variables `$mus1` et `$mus2` avec des tempos à 60 et à des périodes de mise à jour correspondant à 1 temps.

```
@tempo_var $mus1(60,1)  $mus2(60,1)

group G1 @target[5s] @sync $mus1 { ... }

group G2 @target[2s] @sync $mus2 { ... }
```

On suppose ici que les mises à jour sont réalisées depuis l'environnement extérieur où deux musiciens transmettent leur pulsation via un contrôleur. Le groupe d'actions G1 va alors se synchroniser avec le premier musicien, G2 avec le second.

On peut également être à l'écoute de rythmes plus compliqués en modifiant la position courante de la variable au moment des mises à jour. Nous travaillons actuellement sur l'ajout de nouvelles structures permettant de spécifier ce genre de sous-partitions qui intègrent l'ap-

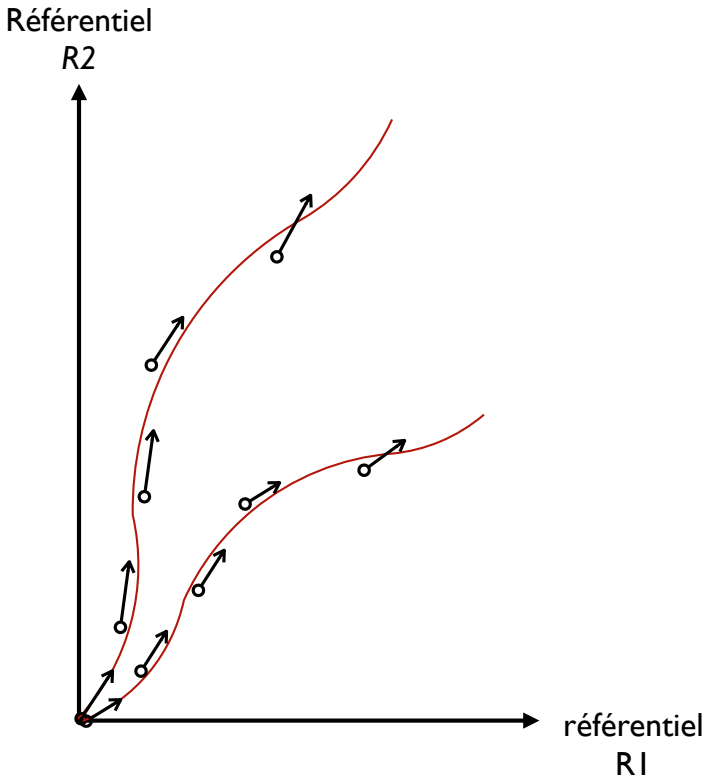



FIGURE 26: Graphiques représentant la temporalité de deux processus musicaux, chacun étant coordonné indépendamment sur la mise à jour de deux variables.

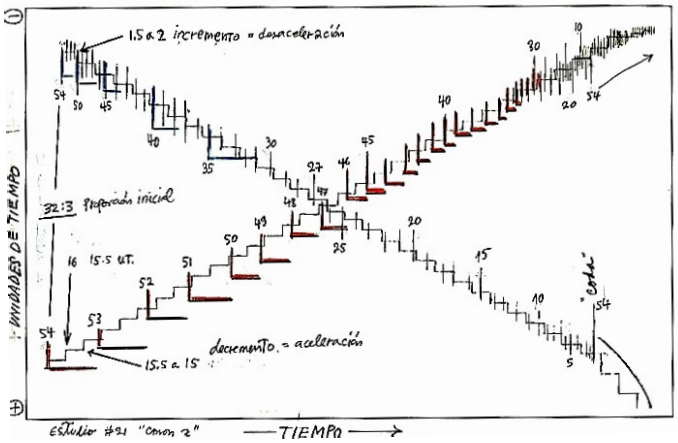



FIGURE 27: Description graphique de la pièce Study 21, *Canon X*, du compositeur C. Nancarrow représentant la relation entre les tempos de deux voix périodiques. Les abscisses correspondent au temps, les ordonnées à la valeur de tempo

proche probabiliste de la machine d'écoute, pour être par exemple, tolérant aux erreurs de mises à jour.

## 6.6 TEMPO CALCULÉ EXPLICITEMENT

Les compositeurs ont l'habitude de manipuler dans leurs partitions des échelles temporelles différentes qui s'exécutent en parallèle. Le compositeur Colon Nancarrow a étudié et mis en pratique cette idée de superposition de tempos notamment dans une de ses études pour piano mécanique, appelée *Canon X*. La figure 27 montre le diagramme original représentant l'organisation temporelle des deux voix constituant la pièce ; la première accélère pendant que la deuxième décélère.

Le langage d'*Antescofo* permet la spécification de ce genre de processus musicaux. Dans l'exemple décrit ci-dessous, il s'agit d'exécuter des processus avec des comportements temporels qui dépendent du jeu du musicien et d'expressions évaluées dynamiquement. La partition suivante décrit deux séquences qui se répètent, l1 et l2, dont l'expression des tempos dépend du tempo estimé du musicien et respectivement de $r1 et $r2, deux paramètres dont l'évolution est spécifiée à l'aide de constructions `curve`. Ces variables varient de manière quasi continue (avec un pas de 0.1 seconde) de 1/4 à 4 pour $r1 et de 4 à 1/4 pour $r2. Ainsi l1 accélère tandis que l2 ralentit pendant une durée de 40 secondes. Notons ici l'utilisation combinée du temps physique (en secondes) et du temps relatif (en pulsations). La figure 28 montre l'évolution des tempos des deux séquences dans le cas d'une interprétation musicale.

```
curve @grain=0.1s {
```

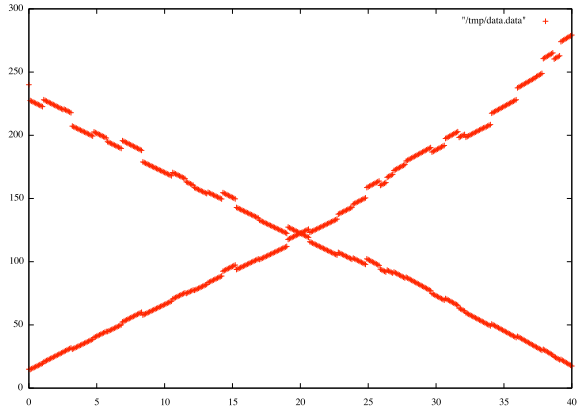



FIGURE 28: Courbes des tempos de deux voix périodiques contrôlées par Antescofo et dépendantes du tempo estimé du musicien. Les abscisses correspondent au temps, les ordonnées à la valeur de tempo.

```
    $r1,$r2{
               [1/4, 4]
       40s  [4, 1/4]
    }
  }
  loop l1 2.0 @tempo=$RT_TEMPO*$r1 {
    actions...
  }
  loop l2 2.0 @tempo=$RT_TEMPO*$r2 {
    actions...
  }
```

### 6.6.1 *Récursivité temporelle*

Dans cette section nous nous intéressons à la notion de récursivité temporelle supportée par le langage d'*Antescofo*. Il est parfois plus simple de penser et de décrire un processus musical de façon récursive. Cette notion de récursivité temporelle est introduite dans les langages *Impromptu* et *ExTempore* dédiés au « live-coding » et construits sur le langage *Scheme*.

```
@proc_def ::CANON($factor,$tempo){
  group @tempo := $tempo {
   loop 5 {
       synth 59
    1. synth 59+(2*$factor)
    1. synth 59-(1*$factor)
    1. synth 59+(1*$factor)
    1. synth 59-(2*$factor)
   }
   $next_factor := $factor * @rand(0.9,1.1)
   $next_tempo := tempo * @rand(0.9,1.1)
```

```
   $del :=  @rand(5)

   $del ::CANON($next_factor, $next_tempo)
  }
 }
::CANON(1, 120)
20s abort ::canon
```

À la manière de Ligeti dans *Kammerkonzert*, qui définit différents *ambitus* et vitesses d'un même motif pour plusieurs instruments de l'orchestre afin de créer des effets de texture, nous définissons le processus récursif `::CANON`. Celui-ci consiste à jouer en boucle une séquence mélodique et à relancer une nouvelle séquence en utilisant le principe de récursivité temporelle. L'ambitus et le tempo seront proches de la séquence appelante mais avec de légères variations aléatoires. L'entrée de la nouvelle séquence est également définie de manière aléatoire (`$del` temps après le début de la séquence appelante). Ce processus est lancé la première fois au début de la partition et crée au fur et à mesure une polyphonie de plus en plus grande.

# 7

# UNE SÉMANTIQUE DÉNOTATIONNELLE DE TOUT LE LANGAGE

Dans ce chapitre nous présentons une sémantique de trace du langage d'*Antescofo*, dans un style dénotationnel. Notre objectif à long terme est de s'appuyer sur cette sémantique pour développer de nouveaux mécanismes de synchronisation avec des agents humains, de prouver la correction de certaines transformations (par exemple à des fins d'optimisation), de permettre l'établissement d'invariants qui pourraient être utiles aux compositeurs (e. g., pour s'assurer que certaines relations temporelles restent vraies quelques soient les variations de l'interprétation du musicien), et de développer une notion d'observation et d'équivalence permettant de tester le système *Antescofo*.

Ce chapitre s'organise comme suit :

- La section suivante (§7.1) introduit la construction de domaine nécessaire à la formalisation d'une *séquence temporisée*. Cette construction nous permet de définir la notion de *trace temporisée* qui servira à représenter la trace d'exécution d'un programme mais aussi la notion d'environnement (§7.2).
- Un programme *Antescofo*, dans une syntaxe abstraite, est formalisé par la notion de *programme temporisé*, qui est aussi un cas particulier des séquences temporisées (§7.3). La section 7.4 entre dans le détail de la représentation d'un programme *Antescofo* dans cette syntaxe abstraite et donne des exemples de cette syntaxe abstraite.
- Nous définissons précisément les fonctions auxiliaires dont nous aurons besoin pour définir les équations sémantiques spécifiant par cas la fonction $\mathbf{Eval}[\![\, \mathbf{P}, \mathbf{R}, \overline{\mathbf{R}} \,]\!]\rho$ qui calcule la trace d'exécution d'un programme (§7.7).
- La section 7.8 détaille le calcul du tempo pour les stratégies de synchronisation anticipative.
- Quelques propriétés élémentaires de cette sémantique sont ensuite présentées (§7.9).

Mais, dans le reste de cette introduction, nous préciserons la place de la sémantique que nous présentons ici. En effet, plusieurs sémantiques formelles du langages ont été développées. Nous précisons ensuite l'approche de modélisation que nous avons suivi.

LES SÉMANTIQUES D'ANTESCOFO. Dans [ECGJ11] nous avons développé une première sémantique de trace, aussi dans un style dénotationnel. Cette première tentative présentait plusieurs inconvénients :

le traitement du temps était « externalisé » sous la forme de l'hypothèse de l'existence d'une fonction de datation [1]. Si cette approche est pertinente pour la modélisation du musicien et de l'environnement extérieur, elle ne l'est plus quand on veut définir la sémantique de temporalités complexes définies par un calcul interne. Par ailleurs la structure des programmes était aplaties, ce qui complique la gestion des erreurs (modélisation des comportements `@local` et `@global` d'une action).

Pour pallier à ce dernier problème, mais surtout dans la perspective de de s'appuyer sur des outils existants permettant la génération automatique de test, nous avons développé par la suite une modélisation des programmes *Antescofo* sous la forme d'un *automate temporisé* [EJCG13]. Cette sémantique ne permet de modéliser que la partie la plus statique d'un programme *Antescofo* : pas d'expression dans les programmes (tous les délais sont constants), pas de constructions comme les `loop`, `whenever`, `forall`, `processus`..., et pas de constructions temporelles complexes : toutes les durées ne doivent faire référence qu'à un seul temps (soit relatif soit physique). Cette sémantique ne permet donc pas d'étudier la création et la gestion de repères temporels qui est une des caractéristiques d'*Antescofo*. En contrepartie, cette approche offre des outils permettant de définir une notion de tests et de couverture de tests (par exemple en utilisant UPPAAL et COVER). Cette approche s'est révélée fructueuse et un système de test automatique de la partie statique du langage est en cours de développement [PSJ14, PJ15]. L'article [EJCG13] détaille le codage d'un fragment d'*Antescofo* sous la forme d'un automate temporisé.

La sémantique présentée ici est donc la troisième étude effectuée pour formaliser le langage. L'objectif de cette étude est d'avoir une description formelle *complète* : en particulier, pour la première fois nous spécifions complètement le traitement du tempo dans le langage.

Bien que la question soit légitime, nous ne montrerons pas que la sémantique en terme d'automates temporisés et la semantique présentée ici sont équivalentes, notre motivation étant principalement de prendre en compte le calcul dynamique qui n'est pas traité dans la première.

Le style dénotationnel développé ici est particulièrement effectif : nous en dérivons directement en annexe un interprète en OCAML. Cet interprète traduisant directement les équations sémantiques, il peut servir d'oracle, y compris pour les programmes incorrects. Cependant cet interprète en OCAML ne s'exécute pas « en temps réel » : les entrées du programme et les traces d'exécution ne sont pas produites « dans le temps » mais en mémoire, sous la forme d'une liste.

1. Cette voie est aussi celle suivie par [BJMP13] qui modélise le même fragment dans un système de règle de déduction naturelle. Ces règles se traduisent presque directement dans un interprète en ReactiveML.

MODELISATION. Le langage d'*Antescofo* va au-delà de la manipulation de séquences d'événements atomiques tels qu'ils peuvent être gérés dans les réseaux de Khan par exemple, car il faut rendre compte de l'écoulement « métrique » du temps. Il est donc nécessaire de formaliser non seulement *quelle* information est produite mais aussi *quand* elle est produite et la modélisation de ce *quand* va au-delà d'un rang dans une suite de valeurs.

*séquence temporisée et opérateur ⋆*

Cela nous a conduit à formaliser la notion de *séquence temporisée* qui représente une succession d'événement et de passages du temps. Cette notion n'est pas nouvelle [ACM02] : on retrouve en partie l'approche des langages synchrones : un événement est atomique, il intervient à un instant du temps et ne consomme pas de temps. L'écoulement « métrique » du temps est explicitement modélisé par des éléments dans la séquence temporisée qui indiquent de « combien avance le temps ». Mais, contrairement à *Chuck* qui produit ces éléments via l'opérateur `+=>`, ces éléments résultent dans *Antescofo* de l'évaluation en parallèle de multiples délais.

Nous définissons la notion de séquence temporisée à travers la définition d'un opérateur $\star$. L'idée est de construire le domaine correspondant à partir de l'ensemble des événements et de l'ensemble des passages du temps. Cet opérateur abstrait sera utilisé pour construire le domaine des traces mais aussi le domaine des programmes.

*causalité et passage du temps*

Expliciter le passage du temps indépendemment de l'occurrence des événements est très utile pour modéliser des actions qui se produisent dans le même instant mais « dans le bon ordre ». C'est une contrainte que le compositeur rencontre naturellement. par exemple, un filtre audio doit être allumé avant de recevoir ses paramètres de contrôle. Mais il n'est pas naturel de devoir expliciter un délai pour séparer les deux actions : tout délai, aussi petit soit-il, convient. Dans *Antescofo*, le compositeur peut simplement se faire succéder ces actions, sans délais : elles se produiront dans le même instant mais dans le bon ordre.

On voit là un usage du temps qui est lié à la notion de *causalité* : une cause précède ses effets, mais ne nécessite pas nécessairement que du temps passe. La sémantique d'*Antescofo* permet de distinguer clairement ce qui dépend de la causalité dans le calcul de ce qui fait passer le temps.

*trace temporisée*

Une *trace temporisée* représente l'observation de l'exécution d'un programme *Antescofo*. Cette observation peut se réduire à une séquence d'affectations qui se produisent en réaction à une entrée, entrelacée par des écoulements du temps où rien d'autre ne se passe. Les entrées d'un programme et les interactions avec l'environnement extérieur, peuvent eux aussi se représenter par des affectations qui correspondent aux informations décodées par la machine d'écoute à partir du flux audio, et aux interactions avec l'environnement (que ce

soit une communication OSC, la mise à jour externe d'une variable, la réception d'un message Max ou PureData, etc.).

*programme temporisé*

Dans sa représentation abstraite, un programme *Antescofo* est une expression dont l'organisation est aussi structurée comme une séquence temporisée : les événements correspondant sont des actions à évaluer et les passages du temps sont des expressions de délais à calculer. C'est pourquoi nous appelons *programme temporisé* un programme *Antescofo* dans sa forme abstraite. Le résultat de l'évaluation d'un programme temporisé est une trace temporisée et on peut noter qu'une trace temporisée est un programme temporisé correspondant à une constante, tout comme une valeur est une expression.

*fonction d'évaluation*

La fonction permettant d'évaluer un programme temporisé en une trace temporisée est notée :

$$\mathbf{Eval}\,[\![\,\mathbf{P},\mathbf{R},\overline{\mathbf{R}}\,]\!]\,\rho\,.$$

Cette fonction prend quatre arguments : **P** représente l'ensemble des coroutines du programme en cours d'exécution ; **R** correspond à l'ensemble des réactions actives qui attendent un événement pour créer une coroutine ; l'argument $\overline{\mathbf{R}}$ correspond à un sous-ensemble des réactions actives qui sont momentanément inhibées (une astuce technique pour gérer dynamiquement les court-circuits temporels [CP01]) ; et le dernier argument $\rho$ représente l'état mémoire du programme [2].

*une machine chimique « temporisée »*

La fonction **Eval** est définie par cas ce qui en fait la solution d'une équation de point-fixe qui formalise le déclenchement d'une réaction de $\mathbf{R}\backslash\overline{\mathbf{R}}$ à un élément de **P**. L'élément déclencheur est retiré de **P** et le résultat de la réaction y est rajouté. Ce schéma rappelle la sémantique du langage *GAMMA* et la machine chimique [BB90] où un ensemble de réactions agit sur un multi-ensemble de données. La différence ici est que les réactions sont « temporisées », qu'elles agissent sur un ensemble de coroutines et non un multi-ensemble de données et que l'on mémorise chaque changement de l'état du programme dans une trace.

## 7.1 CONSTRUCTION DES DOMAINES

Soit $\mathsf{E}$ un ensemble d'événements et $\mathsf{D}$ un ensemble de durées. Pour le moment $\mathsf{E}$ et $\mathsf{D}$ restent abstraits, on ne s'intéresse pas à leur interprétation. Nous voulons définir un monoïde $\mathsf{E}\star\mathsf{D}$, i. e. un ensemble muni d'une opération associative $\bullet$ et d'un élément neutre $\varepsilon$, qu'on appelera *séquence temporisée* et qui représentera les successions finies et infinies d'événements et de passages du temps. L'opération $\bullet$ représente la succession. Afin d'assurer l'existence d'une solution cal-

2. L'état mémoire du programme associe une valeur à chaque variable. On parle d'*environnement*. Cette notion n'est pas à confondre avec la notion d'environnement « extérieur » qui correspond au musicien et plus généralement à toutes les entrées/-sorties du programme.

culable à toute équation de point-fixe sur $E \star D$, nous demanderons à ce que cette construction soit un domaine [GS90].

Un programme *Antescofo* est séquentiel : si à un certain point le calcul se passe mal (erreur dans l'évaluation d'une expression, évaluation qui boucle), alors le calcul tout entier se passe mal à tout instant ultérieur. Par ailleurs, nous voulons distinguer les programmes qui se terminent en un temps fini, des calculs infinis qui ne produisent qu'une trace finie. Cela nous amène à définir $E \star D$ comme la plus petite solution de l'équation

$$E \star D = (E_\perp \oplus D_\perp)^* \oplus (E_\perp \oplus D_\perp)^\$ .$$

Nous suivons fidèlement les notations introduites dans [Mos90] :

- Si $X$ est un domaine, sa relation d'ordre est notée $\sqsubseteq_X$ et le plus petit élément, l'élément indéfini, est noté $\perp_X$. Pour alléger les notations, l'indication du domaine en indice est souvent omis.
- Si $X$ est un ensemble, le *domaine plat* $X_\perp$ est formé de l'élément $\perp_X$ et des éléments de $X$, où tous les éléments de $X$ sont incomparables entre eux et tous plus grand que $\perp_X$.
- Si $X$ et $Y$ sont des domaines, le domaine $X \oplus Y$ dénote la *somme amalgamée* qui contient une copie distinguée des éléments de $X$ et de $Y$, sauf pour l'élément $\perp_X$ qui est identifié avec $\perp_Y$. Les (autres) éléments issus de $X$ sont incomparables aux éléments de $Y$. On peut remarquer que le caractère plat est conservé par cette construction : si $X$ et $Y$ sont des domaines plats, $X \oplus Y$ est un domaine plat. Pour alléger les notations, nous omettrons systématiquement les injections entre les domaines $X$ ou $Y$ est le domaine $X \oplus Y$.
- Le *produit amalgamé* $X \otimes Y$ de deux domaines $X$ et $Y$ est défini en identifiant dans le produit cartésien $X \times Y$ tous les couples dont un éléments est $\perp$. Le caractère plat est conservé par cette construction.
- Si $X$ est un domaine, le domaine $X^*$ représente le domaine des *séquences finies* d'éléments différents de $\perp_X$. Les séquences de longueur différentes ne sont pas comparables et $x_1 \dots x_p$ est inférieur à $x'_1 \dots x'_p$ si et seulement si $x_i$ est inférieur à $x'_i$ pour $1 \leqslant i \leqslant p$.
  Le domaine $X^*$ est isomorphe à
  $$X^* \simeq \{\varepsilon\} \oplus (X \otimes X^*) .$$
  L'élément $\varepsilon$ représente la séquence vide. Nous écrirons
  $$x_1 \bullet x_2 \bullet \cdots \bullet x_n$$
  pour les séquences finies d'éléments $x_i \neq \perp_X$.
- Le domaine $X^\$$ spécifie les *séquences partielles infinies* d'éléments du domaine $X$ où un élément $\perp_X$ ne peut être suivi par des éléments qui ne sont pas $\perp_X$. Ce domaine est isomorphe à
  $$X^\$ \simeq X \otimes X^\$ .$$

Ce domaine rend compte de la séquentialité : si le calcul se passe mal à un moment donné et prend la valeur $\bot_X$, alors à tout instant ultérieur, le calcul se passe mal et prend la valeur $\bot_X$.
L'ordre sur le domaine $X^\$$ est défini de la manière suivante : $x \sqsubseteq_{X^\$} x'$ si chaque élément de la séquence infinie $x$ approxime l'élément correspondant de $x'$. Par suite, $\bot_{X^\$}$ correspond à une séquence infinie de $\bot_X$.
Nous écrirons

$$x_1 \bullet x_2 \bullet \cdots \bullet x_n \bullet \bot$$

pour une séquence infinie dont le premier élément indéfini apparaît au rang $n+1$. Il faut noter que les éléments dénotés par $x_1 \bullet \cdots \bullet x_n \bullet \bot$ ne représente qu'un sous-ensemble de $X^\$$.

Comme nous omettons de noter les injections des domaines de départ dans une somme amalgamée, les éléments de $E \star D$ correspondant à des traces finies se notent directement par $x_1 \bullet \cdots \bullet x_p$ et $x_1 \bullet \cdots \bullet x_q \bullet \bot$.

L'ordre usuel sur $E \star D$ induit par les constructions de domaine rend incomparable les séquences finies et les séquences infinies. Cela implique de choisir au début du programme, avant de lire la première entrée, si le résultat va être fini ou infini, ce qui n'est pas suffisamment expressif : un calcul qui est soit fini soit infini suivant son entrée [3] ne sera pas continu. C'est pourquoi on munit $E \star D$ de la relation d'ordre $\sqsubseteq$ suivante :

| | |
|---|---|
| $\bot \sqsubseteq x$ | pour tout $x \in E \star D$ |
| $x_1 \bullet \cdots \bullet x_p \bullet \bot \sqsubseteq x_1 \bullet \cdots \bullet x_q$ | pour tout $0 < p \leqslant q$ et $x_i \in E \cup D$, |
| $x \sqsubseteq y$ | pour tous les $x, y \in (E_\bot \oplus D_\bot)^\$$ tels que $x \sqsubseteq_{(E_\bot \oplus D_\bot)^\$} y$. |

Le diagramme de Hasse de cette relation est dessiné à la figure 29.

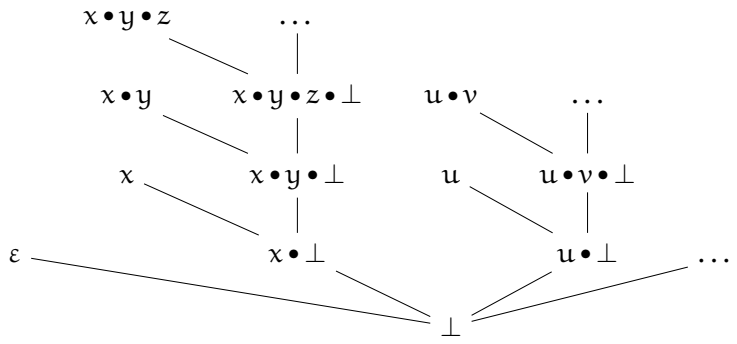



FIGURE 29: L'ordre $\sqsubseteq$ défini sur $E \star D$.

3. Par exemple la fonction qui répète en sortie une séquence d'entrées booléennes jusqu'à ce que cette entrée soit vraie.

OPÉRATION DE SUCCESSION DANS $E \star D$ L'opération de succession $\bullet$ dans $E \star D$ est définie de la manière suivante :

$$\begin{aligned}(x_1 \bullet \cdots \bullet x_p) \bullet (y_1 \bullet \cdots \bullet y_q) &= x_1 \bullet \cdots \bullet x_p \bullet y_1 \bullet \cdots \bullet y_q \\ (x_1 \bullet \cdots \bullet x_p) \bullet (y_1 \bullet \cdots \bullet \bot) &= x_1 \bullet \cdots \bullet x_p \bullet y_1 \bullet \cdots \bullet \bot \\ s \bullet s' &= s, \text{ pour tout } s \in (E_\bot \oplus D_\bot)^\$ \text{ et } s' \in E \star D\end{aligned}$$

pour $x_i, y_j \in E \cup D$. L'opération $\bullet$ est *associative* et $\varepsilon$ (la séquence vide) est l'élément neutre. Notons que $\bot$ n'est pas un élément neutre : il est absorbant à droite. L'opération $\bullet$ est une fonction continue de ses deux arguments. Ce n'est pas une fonction stricte. Cependant, si un de ses arguments est partiel, le résultat est partiel.

SÉQUENCE TEMPORISÉE INFINIE ET PARTIELLE. La construction $\star$ diffère des séquences d'événements temporisées étudiées dans [ACM02]. Dans ce travail, les séquences temporisées sont définies comme un produit quotienté du monoïde $E$ et des passages du temps $D$ afin de ne garder que des alternances finies de $E^*$ et $D$ : $e \bullet d \bullet d' \bullet e'$ est équivalent à $e \bullet (d + d') \bullet e'$. Cela diffère de deux manières par rapport à la construction proposée ici. Notre approche permet de représenter des séquences infinies et nous n'agrégeons pas les passages du temps consécutifs.

Cette dernière propriété n'est en effet pas pertinente dans notre contexte : attendre $d + d'$ après un événement $e$ n'est pas la même chose qu'attendre $d$ après $e$ puis attendre $d'$. En effet, l'évaluation de $d'$ peut être indéfinie et, dans le premier cas, le programme devient indéfini après l'occurrence de $e$, alors que dans le second cas, il devient indéfini après avoir attendu $d$ unités de temps après l'occurrence de $e$.

Ne pas agréger les passages consécutif du temps permet de distinguer les deux programme de la figure 30 même s'ils ne produisent aucune trace de sortie : le processus `P` attend une pulsation puis attend une pulsation, indéfiniment. Le processus `Q` attend deux pulsation puis deux pulsations, indéfiniment. Les traces des deux processus sont différentes : $1 \bullet 1 \bullet 1 \bullet 1 \cdots$ et $2 \bullet 2 \bullet 2 \bullet 2 \cdots$.

Par ailleurs, il est bien pertinent de distinguer la séquence $d \bullet \bot$ de $\bot$. Examinons en effet le programme :

$exp_1$ `$x :=` ` `$exp_2$

```
@proc_def ::P()
{
    1 ::P()
}
::P()
```

```
@proc_def ::Q()
{
    2 ::Q()
}
::Q()
```

FIGURE 30: Deux processus infinis qui ne produisent aucune trace de sortie et qui ne sont pas équivalents.

qui attend simplement un délai qui résulte de l'évaluation de $exp_1$ avant d'évaluer $exp_2$ pour réaliser une affectation. L'évaluation peut être indéfinie pour deux raisons : soit parce que l'évaluation de $exp_1$ est indéfinie, soit parce que l'évaluation de $exp_2$ est indéfinie. Dans le premier cas la trace du programme est $\perp$ alors que dans le second cas, la trace est $d \bullet \perp$ où $d$ est le résultat de l'évaluation de $exp_1$.

Les trois exemples précédents tournent autour de la question de l'observabilité de l'attente d'un délai. Le passage du temps est *a priori* non observable : on peut observer les événements qui bornent ce passage, mais non le passage lui-même. Cependant, attendre un délai dont la durée est bien définie est un comportement observable. On peut en effet imaginer qu'un programme émet dans la trace de sortie un événement distingué pour signifier le début d'une attente et la fin de cette attente. L'attente elle-même ne peut se passer mal : les événements de début et de fin d'attente sont successifs et l'un n'apparaît pas sans l'autre.

Une conséquence de cette modélisation est que nous avons explicitement définit l'opération $\bullet$ pour que les éléments partiels soient absorbants à gauche, ce qui implique que faire $x$ après un nombre infini de délais dont la somme est bornée, comme par exemple

$$1 \bullet \frac{1}{2} \bullet \frac{1}{4} \bullet \cdots \bullet \frac{1}{2^n} \cdots$$

revient à ne jamais faire $x$. En effet, même si la somme $\sum \frac{1}{2^n}$ est finie, chaque délai correspond à un calculs et un nombre infini de calcul ne peut être effectué en un temps fini.

Dan la suite, nous utilisons la construction $\star$ de deux manières : pour construire la trace résultant de l'exéution d'un programme *Antescofo* et pour spécifier les domaines sémantiques dans lesquels les constructions syntaxiques abstraites d'*Antescofo* sont représentées.

## 7.2 TRACES TEMPORISÉES

L'ensemble des *traces temporisées* $\mathcal{T}$ est défini par :

$$\mathcal{T} = \mathcal{U}_{cste} \star \mathbb{Q}^+$$

Un élément de $\mathcal{U}_{cste}$ est un couple $(x, v)$ où $x \in \mathcal{V}ar$ est un identificateur de variable et $v \in \mathcal{V}al$ une valeur, que l'on notera $(x := v)$. Les délais sont représentés par des nombres rationnels positifs (inclus $0$).

TRACE TEMPORISÉE COMME ENVIRONNEMENT. Un élément de $\mathcal{T}$ peut être vu comme un environnement, c'est à dire comme une fonction $\mathcal{V}ar \to \mathcal{V}al$, definie par les équations :

*Valeur d'une variable dans une trace*

$$\begin{aligned} \epsilon(x) &= \bot \\ (s \bullet d)(x) &= s(x) \\ \big(s \bullet (x := v)\big)(x) &= v \\ \big(s \bullet (y := v)\big)(x) &= s(x) \quad \text{where } y \neq x\,. \end{aligned}$$

où $d \in \mathbb{Q}^+$ et $s \in \mathcal{T}$. Autrement dit, la valeur d'une trace $s$ en un identificateur $x$ est la valeur donnée à $x$ par l'affectation la plus récente, ou bien $\bot$ si $x$ n'a pas été assigné ou si $s$ est une séquence infinie.

La mise à jour d'un environnement $\rho$ pour prendre en compte l'affectation $(x := v)$ consiste simplement en une concaténation à droite. Par commodité, on définit la mise à jour incrémentale :

$$\rho \bullet (x \mathrel{+}= v) = \rho \bullet \big(x := \rho(x) + v\big)\,.$$

## 7.3 PROGRAMME TEMPORISÉ ET COROUTINES

Une *coroutine* $p_\tau \in \mathcal{P}_\Upsilon$ correspond à un programme $p$ en cours d'évaluation dans un certain contexte temporel $\tau$. Le *programme temporisé* $p \in \mathcal{P}$ est une séquence temporisée d'éléments de $\mathcal{A}$ qui représentent des actions et d'expressions symboliques de $\mathcal{D}$ qui représentent un délai. Le *contexte temporel* $\tau \in \Upsilon$ est un environnement qui contient les informations nécessaires à la gestion des synchronisations et des relations temporelles lors de l'évaluation, comme par exemple le tempo qui règle l'avancement du temps dans la coroutine. Plus précisément :

$$\begin{aligned} \mathcal{P}_\Upsilon &= \mathcal{P} \times \Upsilon && \text{(\textit{Coroutines})} \\ \mathcal{P} &= \mathcal{A} \star \mathcal{D} && \text{(\textit{Programmes})} \\ \mathcal{A} &= \mathcal{U} \oplus \mathcal{R} \oplus \mathcal{K} \oplus (\mathcal{P} \otimes \Upsilon) && \text{(\textit{Actions})} \\ \mathcal{D} &= \mathcal{E} && \text{(\textit{Délais})} \\ \Upsilon &= \mathcal{V}ar \to \mathcal{E} && \text{(\textit{Contextes temporels})} \end{aligned}$$

où tous les ensembles à l'exception de $\mathcal{P}$, $\Upsilon$ et $\mathcal{P}_\Upsilon$ sont vus comme des domaines plats.

*Notations*

Dans la suite, nous utiliserons les lettres $p, p', q, q', s \ldots$ pour désigner des éléments de $\mathcal{P}$ et $P, Q \ldots$ pour les éléments de $\mathcal{P}_\Upsilon$ ; $a, a', e, e' \ldots$ pour des éléments de $\mathcal{A}$ ; $\tau, \tau', \tau_1, \tau_2 \ldots$ sont des contextes temporels ; les lettres $d, d' \ldots$ dénotent des durées de $\mathcal{D}$ ; l'ensemble $\mathcal{E}$ est l'ensemble des expressions *Antescofo* et les délais sont des expressions. Par commodité, les valeurs sont des expressions (les expressions constantes). Enfin, nous utiliserons les symboles $\rho, \rho' \ldots$ pour désigner des traces (des environnements qui associent des valeurs aux identificateurs de variables).

$\Upsilon$ : CONTEXTE TEMPORELS. Un contexte temporel spécifie les paramètres nécessaires à la gestion des synchronisations et de l'avancement du temps d'une coroutine. Nous représentons ces informations sous la forme d'une environnement qui associe un expression *Antescofo* à des identificateurs prédéfinis :

`uid` : l'identifiant unique d'un programme en cours d'évaluation ;

`tempo` : l'expression *Antescofo* qui définit le tempo de l'évaluation ;

`sync` : la stratégie de synchronisation de la coroutine (`tight`, `loose`, `target`, etc.),

`scope` : la spécification du comportement à suivre en cas d'événement manqué (`global` ou `local`),

*beatpos* : la position courante en pulsation dans le référentiel temporel de la coroutine ;

`expEvt` : la position de l'événement musical attendu (ou bien *undef* si un tel événement n'est pas attendu) ;

`labels` : un ensemble de labels.

Contrairement aux environnement $\rho$ qui lient une valeur à une variable, nous utilisons une notation pointée pour accéder à ces informations. Par exemple, $\tau$.`tempo` peut retourner l'expression `$RT_tempo`, la variable utilisée par la machine d'écoute pour communiquer le tempo détecté dans le flux audio. A l'exception de `tempo`, toutes les autres informations correspondent à des valeurs.

On suppose que chaque contexte temporel $\tau$ possède un identificateur unique $\tau$.`uid` dont la valeur appartient à un domaine plat $\mathcal{I}d \subset \mathcal{V}al$. En plus de la relation d'ordre du domaine, les éléments de $\mathcal{I}d$ sont totalement ordonnés par la relation d'ordre totale $\prec$ qui représente une priorité utilisée pour choisir quelle coroutine il faut évaluer quand plusieurs coroutines sont prêtes à s'exécuter.

Nous supposons l'existence d'un mécanisme FRESH qui génère un identificateur « frais », c'est-à-dire un identificateur qui n'est pas déjà utilisé par le système.

Un contexte temporel est une fonction partielle sur un domaine fini. On écrira $[x_1 \to exp_1, \ldots, x_n \to exp_n]$ la fonction qui associe $exp_i$ à l'identificateur $x_i$.

Si $\tau_1$ et $\tau_2$ sont deux contextes temporels, $\tau_3 = \tau_1\tau_2$ est le contexte temporel tel que $\tau_3.x$ renvoie $\tau_2.x$ si $x$ est dans le domaine de $\tau_2$ et $\tau_1.x$ sinon.

Pour simplifier les notations, on écrira

$$\tau + d \quad \text{pour} \quad \tau[\mathit{beatPos} \to \tau.\mathit{beatPos} + d]$$

$\mathcal{D}$ : DELAIS DANS UN PROGRAMME TEMPORISÉ. La spécification d'un délai est une expression *Antescofo* dont la valeur définit la durée du délai. On écrit $d$ pour un élément de $\mathcal{D}$, où $d$ dénote aussi l'expression correspondante dans $\mathcal{E}$. Cette expression s'évalue en un

nombre $\mathrm{eval}_\rho(d) \in \mathbb{Q}$. Mais l'interprétation de ce nombre en une durée effective $\Delta_{\rho,\tau}(d)$ en secondes requière un contexte temporel $\tau$ (en plus de l'environnement $\rho$ qui associe une valeur à chaque variable). La définition de la fonction $\Delta_{\rho,\tau}$ qui évalue une expression de délai en seconde est définie au paragraphe 7.5.4.

$\mathcal{A}$ : ACTIONS Une action est un élément de $\mathcal{A}$, c'est-à-dire l'un des éléments suivant :

- une affectation de variable $(x := exp) \in \mathcal{U}$, où $x \in \mathcal{V}ar$ est $exp \in \mathcal{E}$ est une expression *Antescofo* ;
- la terminaison $(\mathtt{abort}\ \ell) \in \mathcal{K}$ des coroutines étiquetées par un label dans $\ell$ ;
- une coroutine (i. e., un programme temporisé muni de son contexte temporel) qui sera évaluée concurremment aux autres coroutines ;
- une réaction $(exp \xrightarrow{V} p) \in \mathcal{R} = \mathcal{E} \times 2_\perp^{\mathcal{V}ar} \times \mathcal{P}$ qui lance l'exécution du programme $p$ si l'expression $exp$ s'évalue à vrai lors de l'affectation d'une des variables présente dans $V$. Nous notons $2_\perp^A$ pour le domaine plat formé de $\perp$ et des sous-ensembles finis de $A$.

Nous dirons qu'une action $e \in \mathcal{A}$ est *composée* si $e \in \mathcal{P} \otimes \Upsilon$ et sinon elle est qualifiée d'*élémentaire*.

ENSEMBLE DE PROGRAMMES TEMPORISÉS ET ENSEMBLE DE RÉACTIONS. Danas la suite, nous avons besoin de manipuler des ensemble de paires $r_\tau$ formées d'une réaction bien définie $r$ et d'un contexte temporel $\tau$. Nous aurons aussi besoin de manipuler des ensembles de paires $p_\tau$ formées d'un programme bien définis $p$ et d'un contexte temporel bien défini $\tau$. L'ensemble $\boldsymbol{\mathcal{R}_\Upsilon} = 2_\perp^{\mathcal{R}\otimes\mp}$ dénote le domaine plat formé des ensembles fini d'éléments $r_\tau \in \mathcal{R} \otimes \Upsilon$ différents de $\perp$ ; et de manière similaire, nous noterons $\boldsymbol{\mathcal{P}_\Upsilon} = 2_\perp^{\mathcal{P}\otimes\Upsilon}$. On écrira en gras $\mathbf{R}$ et $\mathbf{P}$ pour désigner les éléments de ces domaines.

*Nota bene :* ces ensembles représentent un ensemble de réactions simultanément actives ou bien un ensemble de coroutines qui s'exécutent concurremment mais dont toutes doivent arriver à terme. Ils ne représentent pas les différentes exécutions possibles d'une exécution non-déterministe dont une seule sera effectivement réalisée. Nous n'avons donc pas besoin ici d'un domaine représentant l'approximation des sous-ensembles d'un autre domaine (power domain) [GS90] et un domaine plat répond à nos besoins.

ORDRE TOTAL SUR LES PROGRAMMES TEMPORISÉS. Étant donné un ensemble de coroutines, laquelle doit s'évaluer en premier ? Une coroutine étant une séquence d'actions et de passages du temps, celle qui doit s'évaluer en premier est celle qui débute par le plus petit passage du temps (qui peut être une durée de 0). Le problème est un peu plus compliqué : plusieurs coroutine peuvent « être prêtes » en

même temps, i. e. avoir une action à évaluer dans l'instant courant. Nous avons mentionné l'*hypothèse synchrone* qui permet de « lancer deux actions au même instant, mais dans le bon ordre ».

Ce bon ordre provient de critères additionnels. Par exemple, le programmeur s'attend à ce que les actions séparées par un délai de 0 soient exécutées dans l'ordre d'écriture de la partition :

```
$xx := $x + $x
$d := $x * $xx
($d) print OK $d
```

Les trois premières instructions s'exécutent sans délai. Et les langages impératifs séquentiels nous ont habitués à ce qu'elles soient exécutées dans leur ordre d'écriture, ce qui est nécessaire car en général, les dépendances entre variables impératives ne suffisent pas à déterminer l'ordre des calculs. La dernière instruction sera aussi exécutée dans l'instant : la valeur de d après l'évaluation de la troisième action est 0.

Une autre complication provient de la gestion des erreurs : quand les actions marquées comme `@global` sont rattachées à un événement manqué, il faut, au moment de la détection de l'erreur, respecter l'ordre de lancement de ces actions sans tenir compte des délais indiqués, mais en tenant compte de l'ordre induit.

Nous formalisons ces différentes contraintes en spécifiant une *relation d'ordre entre les coroutines*. Cette relation d'ordre permettra de choisir dans un ensemble de coroutines celle qui doit s'évaluer en premier, si on suppose que la relation d'ordre est totale. Comme les délais interviennent dans le choix de cette coroutine, la relation d'ordre dépendra d'un environnement $\rho$. Suite à l'évaluation de la coroutine « minimale », l'environnement peut changer ainsi que l'ensemble des coroutines candidates. Et le choix de la prochaine coroutine à évaluer se fera avec ces nouveaux paramètres.

Nous définissons à présent cette relation d'ordre. Soit $\rho$ un environnement, i. e. une trace temporisée. Les coroutine $p_\tau$ sont partiellement ordonnées par la relation $<_\rho$ induites par les règles suivantes, données en suivant leur ordre de précédence :

$$
\begin{array}{rcll}
\epsilon_\tau & <_\rho & p_{\tau'} & \text{si } p \neq \epsilon \\
p_\tau & <_\rho & p'_{\tau'} & \text{si } \tau.\mathrm{expEvt} \neq \mathit{undef} \text{ et } \tau.\mathrm{expEvt} < \rho(\texttt{\$POS}) \\
 & & & \text{et } (\tau'.\mathrm{expEvt} = \mathit{undef} \text{ ou } \tau.\mathrm{expEvt} < \tau'.\mathrm{expEvt}) \\
(d \bullet p)_\tau & <_\rho & p'_{\tau'} & \text{si } \Delta_{\rho,\tau}(d) = 0 \text{ et } p_\tau <_\rho p'_{\tau'} \\
p_\tau & <_\rho & (d \bullet p')_{\tau'} & \text{si } \Delta_{\rho,\tau'}(d) = 0 \text{ et } p_\tau <_\rho p'_{\tau'} \\
(e \bullet p)_\tau & <_\rho & (d \bullet p')_{\tau'} & \text{si } \Delta_{\rho,\tau'}(d) > 0 \\
(d \bullet p)_\tau & <_\rho & (d' \bullet p')_{\tau'} & \text{si } \Delta_{\rho,\tau}(d) < \Delta_{\rho,\tau'}(d') \\
p_\tau & <_\rho & p'_{\tau'} & \text{si } \tau.\mathrm{uid} \prec \tau'.\mathrm{uid}
\end{array}
$$

pour tous les $p, p'$ dans $\mathcal{P}$, $e, e'$ élément de $\mathcal{A}$, $\tau, \tau'$ de $\Upsilon$ et $d, d'$ dans $\mathcal{D}$. L'évaluation de la fonction $\Delta_{\rho,\tau}$ qui transforme une expression de délais en secondes est définie à la section 7.5.4.

La seconde règle correspond au traitement des erreurs et à l'ordonnancement des actions qui sont rattachées à un événement manqué. Les autres règles traduisent le comportement suivant. Si $P <_\rho Q$, alors soit la première action $e_P$ exécutée P se produit avant la première action $e_Q$ exécutée par Q, ou bien ces actions se produisent dans le même instant, mais l'identifiant du contexte temporel de P est plus petit que l'identifiant du contexte temporel de Q. L'idée est de coder dans la relation d'ordre sur les identifiants de contexte temporels, les règles d'ordonnancement additionnelles évoquées plus haut.

En général $<_\rho$ est un ordre partiel car les identifiants de deux contextes temporels peuvent être égaux. L'ordre $<_\rho$ sur un ensemble de coroutines $\mathbf{P} \in \mathcal{P}_\Upsilon$ sera total si on impose que chaque identifiant est distinct. On peut imposer un critère un peu plus fort, pour simplifier l'établissement de la propriété suivante : toute coroutine issue de l'évaluation des coroutines de $\mathbf{P}$ auront des identifiants uniques. On dira alors que l'ensemble $\mathbf{P}$ est *bien formé*.

Ce prédicat se définit ainsi. Soit $\textsc{uid}(P)$ le multi-ensemble des identifiants des coroutines d'un ensemble $P \in \mathcal{P}_\Upsilon$ :

$$
\begin{aligned}
\textsc{uid}(p_\tau) &= \textsc{uid}(p) \uplus \{\tau.\mathtt{uid}\} \\
\textsc{uid}(\varepsilon) &= \varnothing \\
\textsc{uid}(d \bullet p) &= \textsc{uid}(p) \\
\textsc{uid}(e \bullet p) &= \textsc{uid}(p) && \text{si } e \text{ élémentaire} \\
\textsc{uid}(e \bullet p) &= \textsc{uid}(p) \uplus \textsc{uid}(e) && \text{si } e \text{ composé}
\end{aligned}
$$

où $\uplus$ est l'union des multi-ensembles.

Une coroutine P est dite *bien formée* if P est fini et si $\textsc{uid}(P)$ est un ensemble, i. e. la multiplicité de chaque élément de $\textsc{uid}(P)$ est au plus de 1 (i. e. chaque identifiant est différent). Un ensemble de coroutines $\mathbf{P} \in \mathcal{P}_\Upsilon$ est *bien formé* si chaque P de $\mathbf{P}$ est bien formée et si $\uplus_{P \in \mathbf{P}} \textsc{uid}(P)$ est un ensemble (autrement dit, $\mathbf{P}$ est un ensemble fini et tous les éléments de $\mathbf{P}$ sont finis et tous les identifiants qui apparaissent dans une action d'un élément de $\mathbf{P}$ sont distincts).

Dans la section 7.9 nous montrerons qu'un pas d'évaluation d'une coroutine bien formée induit l'évaluation d'un ensemble bien formé de coroutines. En conséquence, $<_\rho$ est un ordre total sur tous les ensembles de coroutines considérés lors de l'évaluation d'une coroutine bien formée, et l'évaluation est donc déterministe.

TRACE TEMPORISÉ D'ENTRÉE ET PROGRAMME TEMPORISÉ. Les interactions d'un environnement avec un programme *Antescofo*, ses entrées au cours du temps, se formalisent naturellement par du temps qui passe et des mises à jour de variables « externes » qui représentent

l'état de la machine d'écoute (hauteur de la note reconnue, tempo détecté, position dans la partition, etc.). C'est donc une trace temporisée I.

On peut représenter les mises à jours par des affectations : $\mathcal{U}_{\mathtt{cste}} \subset \mathcal{U}$ puisque une valeur est un cas particulier de constante. Les passages du temps dans une trace sont aussi des passages du temps dans un programme temporisé et correspondent à des délais constants de $\mathbb{Q}^+$. Autrement dit, une trace temporisée I est aussi un programme *Antescofo*.

Il suffit donc de munir cette trace d'un contexte temporel $\tau_{\mathcal{I}}$ judicieusement choisi pour en faire une coroutine $I_{\tau_{\mathcal{I}}}$ qui n'aura aucun statut particulier. En particulier, les événements sont des actions atomiques comme les autres.

Le contexte temporel $\tau_{\mathcal{I}}$ est utilisé pour interpréter les informations temporelles d'une trace d'entrée. Ce contexte ne dépend pas de la trace d'entrée. Nous le définissons comme suit :

$$\begin{aligned}
\tau_{\mathcal{I}} = [\quad \mathtt{uid} &\rightarrow -1,\\
\mathtt{labels} &\rightarrow \varnothing,\\
\mathtt{sync} &\rightarrow \mathtt{loose},\\
\mathtt{scope} &\rightarrow \mathtt{global},\\
\mathtt{tempo} &\rightarrow 1,\\
\mathit{beatPos} &\rightarrow 0,\\
\mathtt{expEvt} &\rightarrow \mathit{undef} \quad ]
\end{aligned}$$

L'identifiant $-1 \in \mathcal{I}\mathrm{d}$ est le plus petit des identifiants. Il est réservé au contexte $\tau_{\mathcal{I}}$ ce qui assure que les événements correspondant à une entrée sont traités avant toute autre action qui prendrait place dans le même instant. Les délais dans la trace d'entrée correspondent à l'écoulement du temps physique en seconde. Aussi l'expression $\tau_0.\mathtt{tempo}$ réfère à la constante 1 (cf. la section 7.5.4). Au début des temps, la position en pulsation (*beatPos*) dans la partition est 0 et le prochain événement attendu expEvt est initialisé à *undef* car le programme qui code la trace d'entrée n'attend aucun événement (c'est lui qui les génère).

Représenter une trace d'entrée par le programme qui la génère, a le grand avantage de rendre symétrique le traitement des entrées et des actions du programme. Mais cela restreint notre sémantique à ne spécifier le comportement d'un programme que sur les entrées calculables. Cette restriction n'est qu'apparente. En effet, toute trace finie est évidemment calculable, puisqu'il suffit de la représenter par le programme qui énumère dans le temps ses événements. Les traces infinies qui ne sont pas calculables ne peuvent pas se définir de manière constructive, mais le comportement du programme est bien défini sur tout préfixe fini de cette trace. En effet, un programme *Antescofo* est causal, comme on le verra à la section 7.9, et donc la connaissance

d'un préfixe de la trace d'entrée, même si cette trace n'est pas calculable, fixe un préfixe de la trace de sortie.

## 7.4 REPRÉSENTATION D'UN PROGRAMME *antescofo* PAR UN PROGRAMME TEMPORISÉ

Pour simplifier la sémantique développée ici, nous ne traiterons dans cette étude que les variables globales. La modélisation des variables locales dans la sémantique repose sur des techniques bien connues (comme par exemple gérer une pile d'environnement locaux et mémoriser dans chaque coroutine ses liens de filiation) et ne ferait que compliquer inutilement notre présentation.

### 7.4.1 *Syntaxe abstraite des* `group`

Un programme *Antescofo* est représenté de manière abstraite par un programme temporisé de la manière suivante :

- un `group` et ses attributs de synchronisation sont représentés par un programme temporisé et son contexte temporel (i. e., une coroutine) ;
- la hiérarchie des groupes est rendue par l'imbrication des coroutines ;

Par exemple :

```
group g1 @tight @local
{
   d1 a
   d2 group g2 @loose @tempo=$t
   {
      d3 b
      c
   }
}
```

$\longrightarrow \left(\texttt{d1} \bullet \texttt{a} \bullet \texttt{d2} \bullet (\texttt{d3} \bullet \texttt{b} \bullet \texttt{c})_{\tau_2}\right)_{\tau_1}$

Le contexte temporel associé à un `group` collecte les informations partielles sur la gestion des relations temporelles spécifiées pour ce groupe. Pour le fragment de programme ci-dessus, les environnements sont définis comme :

$$\tau_1 = [\mathit{uid} \to 1, \mathit{labels} \to \{\texttt{g1}\}, \mathit{sync} \to \mathit{tight}, \mathit{scope} \to \mathit{local}]$$
$$\tau_2 = [\mathit{uid} \to 2, \mathit{labels} \to \{\texttt{g1}, \texttt{g2}\}, \mathit{sync} \to \mathit{loose}, \mathit{tempo} \to \texttt{\$t}]$$

Cet exemple montre que les labels sont hérités statiquement des groupes englobants. Toutes les actions à un même niveau partagent les mêmes labels. Les informations temporelles sont partielles : elles seront complétées à l'exécution par des informations dynamiquement héritées du contexte d'exécution. Nous n'associons pas d'environnement temporel au groupe racine `g1`, parce qu'elles dépendent du contexte dans lequel `g1` apparaît.

Les actions atomiques sont directement représentées par des constructions de $\mathcal{A}$ :

- une affectation de variable se traduit en un élément de $\mathcal{U}$ ;
- une commande `abort` se représente par un élément de $\mathcal{K}$ ;
- un `whenever` se convertit en un élément de $\mathcal{R}$ correspondant à une réaction aux mises à jour des variables qui apparaissent dans la garde :

$$\texttt{whenever(\$x>\$y)\{p\}} \quad \longrightarrow \quad (\texttt{(\$x} > \texttt{\$y)} \xrightarrow{\{\texttt{\$x,\$y}\}} \texttt{p})$$

Toutes les autres constructions sont du « sucre syntaxique » [Lan64], comme esquissé ci-dessous.

### 7.4.2 *Syntaxe abstraite des processus*

Une définition de processus se traduit par une réaction et un contexte temporel : la réaction est déclenchée à chaque appel de processus par la mise à jour d'une variable auxiliaire qui sert de déclencheur. Soulignons que le contexte temporel associé à la réaction a pour étiquette le nom du processus mais aucune autre information : les informations manquantes seront héritées lors de l'appel (cf. la définition de $\mathbf{Q}_{\texttt{p}}$ page 122) réalisant ainsi l'héritage des relations temporelles illustré à la section 6.6. Par exemple :

$$\texttt{@proc\_def ::P()\{p\}} \quad \longrightarrow \quad \left(\texttt{\$P\_trigger} \xrightarrow{\{\texttt{\$P\_trigger}\}} \texttt{p}\right)_{[labels \to \texttt{P}]}$$

Un appel de processus se traduit par l'affectation de la variable auxiliaire de déclenchement :

$$\texttt{::P()} \quad \longrightarrow \quad (\texttt{\$P\_trigger := true})$$

### 7.4.3 *Syntaxe abstraite des autres actions*

Toutes les autres constructions *Antescofo* peuvent se coder dans un fragment du langage qui ne contient que des `group` et des `whenever`[4]. Nous illustrerons ici simplement la `loop` et la conditionnelle.

ITERATION. La construction `loop` se traduit par un `whenever` qui est activé par une variable auxiliaire affectée périodiquement :

```
loop $period {p} ⟶
                    whenever($trigger_l)
                    {
                        group {p}
                        ($period) $trigger_l := true
                    }
                    $trigger_l := true
```

4. Notons que ce n'est pas le cas d'un processus qui se traduit directement par une réaction : le contexte temporel associé à la réaction traduisant un processus n'est pas le même que le contexte temporel associé à la réaction traduisant un `whenever`.

CONDITIONELLE. Une conditionnelle se traduit par deux `whenever`, chacun réalisant une des branches de la construction :

```
if (exp)                    whenever ($trigger = true)  { p }
{ p }          ⟶           whenever ($trigger = false) { q }
else                        $trigger := exp
{ q }
```

### 7.4.4 *Syntaxe abstraite des programmes*

Dans notre modélisation, nous supposons que la machine d'écoute fournit une séquence temporisée qui correspond à la mise à jour de la variable `$POS` avec la position (dans le score et en pulsation) de l'événement qu'elle détecte dans le flux audio. Le déclenchement des actions sur l'occurrence d'un événement se traduit alors par un ensemble de `whenever` qui sont actifs simultanément :

```
                            ; 0 is the position of NOTE C4
                            whenever(isUndef($once1) & $POS >= 0) {
                              $once1 := 1
                              $RNOW := $POS
NOTE C4 1.5                   d1 a1
d1 a1                         d2 a2
d2 a2              ⟶       }
NOTE E4 2.0                 ; 1.5 is the position of NOTE E4
a3                          whenever(isUndef($once2) & $POS >= 1.5) {
                              $once2 := 1
                              $RNOW := $POS
                              a3
                            }
```

La variable `$once`i est associée à l'événement `i` de la partition. Elle est utilisée pour s'assurer que le `whenever` chargé du déclenchement des actions associées à l'événement `i` n'est lancé qu'une seule fois. En effet, le prédicat isUndef n'est vrai que si son argument, une variable, n'a jamais été assigné. Ainsi, la garde du `whenever` ne peut être vraie qu'avant la première activation du corps du `whenever`.

La seconde partie de la garde du `whenever` compare la valeur de `$POS` à la position dans le score de l'événement musical. Le `whenever` est déclenché (une fois) dès que cette position dépasse la position de l'événement : un événement manqué peut en effet amener à sauter un événement mais les actions associées doivent quand même être déclenchées, quand celles-ci sont `@global`. Cette situation est détectée en comparant la valeur de la variable `$RNOW` avec la valeur de la position de l'événement attendu $\tau.\mathrm{exp}\mathrm{Evt}$ dans un contexte temporel $\tau$.

Les codages décrits dans cette section permettent de représenter une partition augmentée *Antescofo* $S \in \mathcal{R}_\Upsilon$ par un ensemble de paires

$r^i_{\tau_i}$ où $r^i$ est la réaction correspondant au ième événement musical et $\tau^i$ le contexte temporel associé :

$$\tau^i = \left[\mathtt{uid} \to \textsc{Fresh},\ \mathtt{tempo} \to \mathtt{\$RT\_tempo}, \mathtt{expEvt} \to \mathtt{pos}^i\right]$$

On fera l'hypothèse que $\tau^i.\mathtt{expEvt} < \tau^j.\mathtt{expEvt}$ pour $i < j$. Le contexte temporel $\tau^i$ se complétera lors de l'évaluation avec les informations provenant de $\tau_0$ pour déterminer le contexte temporel des actions déclenchées « à la racine ». La variable `$RT_tempo` est la variable utilisée par la machine d'écoute pour communiquer l'estimation du tempo du musicien. Dans ce contexte temporel, la position en pulsation quand le `whenever` est déclenché, est la position de l'événement musical (même si celui-ci a été manqué, c'est la position de la première action attachée à cet événement). Dans l'exemple précédent, si on suppose que `NOTE C4 1.5` est le premier événement de la partition, alors $\mathtt{pos}^1 = 0$ et $\mathtt{pos}^2 = 1.5$. On note $\mathit{Score}$ l'ensemble des positions $\mathtt{pos}^i$ des événements d'une partition augmentée.

## 7.5 FONCTIONS AUXILIAIRES

Nous définissions sept fonctions auxiliaires utilisées dans la spécification des équations sémantiques au paragraphe 7.7.

### 7.5.1 *Évaluation des expressions*

La fonction $\mathrm{eval} : \mathcal{T} \to \mathcal{E} \to \mathcal{V}\mathtt{al}$ évalue une expression *Antescofo* en utilisant un environnement (sous la forme d'une trace temporisée). Nous ne spécifierons pas plus avant cette fonction, car elle est tout à fait standard.

### 7.5.2 *Sélection de la prochaine coroutine à évaluer*

La fonction

$$\mathrm{next} : \mathcal{T} \to \mathcal{P}_{\boldsymbol{\Upsilon}} \to \mathcal{A} \times \mathcal{P} \times \Upsilon \times \mathcal{P}_{\boldsymbol{\Upsilon}}$$

prend un environnement $\rho$ (une trace temporisée) et un ensemble de coroutines $\mathbf{P}$ et retourne un quadruplet : la première action de la coroutine qui doit s'évaluer en premier, la suite du programme temporisé correspondant à cette coroutine, le contexte temporel de cette coroutine, et le reste des coroutines de $\mathbf{P}$ :

$$\mathrm{next}_\rho(\mathbf{P}) = (x, p, \tau, \mathbf{P}') \quad \text{avec } (x \bullet p)_\tau = \min_{<_\rho}(\mathbf{P}) \text{ et } \mathbf{P}' = \mathbf{P} \backslash \{(x \bullet p)_\tau\}.$$

*continuation d'une coroutine*

L'expression $A \backslash B$ dénote les éléments de $A$ qui n'appartiennent pas à $B$. La séquence temporisée $p$ est appelée la *continuation* de la coroutine $(x \bullet p)_\tau$ La fonction next est bien définie, i. e. un élément minimal existe, si $\mathbf{P}$ est bien formée et si $\mathbf{P} \neq \varnothing$. Si $\mathbf{P} = \varnothing$ ou bien si $\mathbf{P}$ n'est pas bien formé, alors $\mathrm{next}_\rho(\mathbf{P}) = \bot$.

### 7.5.3 *Sélection de l'événement associé à une action* `@tight`

La fonction precScore : $\mathbb{Q} \to \mathbb{Q}$ retourne le plus grande des positions $pos^i$ des événements de la partition qui sont inférieurs à l'argument :

$$\text{precScore}(pos) = \max\{pos' \mid pos' \in Score, pos' \leqslant pos\}$$

où $Score$ désigne l'ensemble des positions $pos^i \in \mathbb{Q}^+$ des événements musicaux de la partition.

La fonction succScore : $\mathbb{Q} \to \mathbb{Q}$ retourne la plus petite des positions qui sont plus grandes que l'argument :

$$\text{sucScore}(pos) = \min\{pos' \mid pos' \in Score, pos' > pos\}$$

### 7.5.4 *Traduction d'une durée relative à un contexte temporel en une durée en seconde*

La fonction $\Delta_{\rho,\tau} : \mathcal{T} \to \Upsilon \to \mathcal{D} \to \mathbb{Q}$ retourne la valeur en seconde de la durée $d$ débutant à la date `$NOW` dans le contexte temporel $\tau$ :

$$\Delta_{\rho,\tau}(d) = dsec$$
$$\text{avec} \quad d' = \int_o^{o+dsec} |f(t)|\,dt, \quad d' = max(0, \text{eval}_\rho(d)) \quad \text{et} \quad o = \rho(\texttt{\$NOW})$$

L'opérateur max et la valeur absolue dans les expressions précédentes permet de s'affranchir de spécification aberrantes comme une durée ou un tempo négatif.

L'expression du tempo $f$ dépend de la spécification $\tau.tempo$. Si $\tau.sync$ spécifie une stratégie de synchronisation statique, i. e. $\tau.sync \neq target\ldots$ alors $f = \text{eval}_\rho(\tau.tempo)$. L'expression du tempo dans le cas des stratégies anticipatives est définie à la section 7.8 et nécessite la connaissance de $\tau.beatPos$, la valeur courante de `$RNOW` et le paramètre (fenêtre ou cible) de la stratégie.

Quelle que soit la stratégie de synchronisation, l'expression du tempo $f$ spécifie une fonction unaire. L'argument de cette fonction est le temps physique $t$. Un cas particulier est la constante $c$ qui est interprétée alors comme la fonction constante $\lambda t.c$.

Le calcul de $dsec$ s'effectue de la manière suivante :

- Pour les tempos constants $c \in \mathbb{Q}$, on a $dsec = d'/c$.
- Pour les stratégies de synchronisation anticipatives, l'expression du tempo permet de calculer $dsec$ analytiquement. Ce calcul est détaillé à la section 7.8.
- Pour les fonctions de tempo arbitraires, on utilise une méthode numérique de Newton-Raphson [Pre07] pour trouver par approximations successives la racine de la fonction $h(x) = d' - \int_o^{o+x} |f(t)|\,dt$. Cette racine est la valeur $dsec$ cherchée.

La méthode de Newton-Raphson nécessite le calcul de la dérivée de $h$ à un point arbitraire $x$, ce qui dans notre cas peut se calculer exactement puisque ce calcul ne demande que le calcul de $f$ au point $x$. L'intégration se fait aussi par une méthode numérique. Nous prenons comme valeur initiale des itérations le point $o$. Les itérations successives sont stoppées dès qu'une valeur suffisamment précise est atteinte (par exemple quand la différence entre deux approximations successives est inférieure à la milliseconde).

### 7.5.5 *Traduction d'une durée en seconde en une durée relative à un contexte temporel*

La fonction inBeats : $\mathcal{T} \to \Upsilon \to \mathbb{Q} \to \mathbb{Q}$ retourne la valeur en pulsation dans le contexte temporel correspondant à $\tau$, de la durée $dsec$ débutant à la date `$NOW` et donnée en seconde :

$$\text{inBeats}_{\rho,\tau}(dsec) = \int_{o}^{o+dsec} |f(t)|\,dt$$

où $o = \rho(\texttt{\$NOW})$ et $f$ est calculée comme dans la définition de la fonction $\Delta$ : $f = \text{eval}_\rho(\tau.tempo)$ si $\tau.sync \neq target$, etc.

### 7.5.6 *Evaluation du délais de tête d'un programme*

La fonction evalFirstDelay : $\mathcal{T} \to \mathcal{A} \star \mathcal{D} \to \mathcal{A} \star \mathcal{D}$ évalue l'expression du premier délai d'un programme, si celui-ci débute par un délai :

$$\text{evalFirstDelay}_\rho(p) = \begin{cases} \text{eval}_\rho(d) \bullet q & \text{si } p = d \bullet q \text{ avec } d \in \mathcal{D} \\ p & \text{sinon} \end{cases}$$

### 7.5.7 *Suppression des actions portant un label donné*

La fonction $\text{erase}_\ell(\mathbf{P})$ supprime de l'ensemble de coroutines $\mathbf{P}$ toutes les coroutines qui portent le label $\ell$ :

$$\text{erase}_\ell(\mathbf{P}) \quad = \quad \{P \mid P = p_\tau \in \mathbf{P} \text{ et } \ell \notin \tau.labels\}$$

| | |
|---|---|
| $(E \star D, \bullet)$ | les séquences temporisées d'événements dans $E$ et de durées dans $D$ muni de l'opération de succession $\bullet$ (p. 101) |
| $\mathcal{V}ar$ | les identificateurs de variables (p. 104) |
| $\mathcal{V}al$ | les valeurs *Antescofo* (p. 104) |
| $\mathcal{E}$ | les expressions *Antescofo* (p. 105) |
| $\mathcal{D}$ | les expressions dénotant un délai |
| $\mathcal{A}$ | les actions (p. 107) |
| $(x := exp) \in \mathcal{U}$ | les actions d'affectation où $x \in \mathcal{V}ar$ et $exp \in \mathcal{E}$ (p. 107) |
| $(x := v) \in \mathcal{U}_{cste}$ | les affectations avec $x \in \mathcal{V}ar$ et $v \in \mathcal{V}al$ : les actions d'affectations dont le membre droit est une constante (p. 110) |
| $(abort\ \ell) \in \mathcal{K}$ | les actions de terminaison (p. 107) |
| $(exp \xrightarrow{V} p) \in \mathcal{R}$ | les actions de réaction : $exp \in \mathcal{E}, V \in 2^{\mathcal{V}ar}_{\perp}, p \in \mathcal{P}$ (p. 107) |
| $\rho \in \mathcal{T}$ | les traces temporisées $\mathcal{U}_{cste} \star \mathbb{Q}^+$ (p. 104) ; une trace temporisée est utilisée comme environnement : $\rho(x)$ où $x \in \mathcal{V}ar$ ; la mise à jour d'un environnement se note avec $\bullet$ |
| $\tau \in \Upsilon$ | les contextes temporels : $\mathcal{V}ar \to \mathcal{E}$ (p. 106) ; sont utilisés les identificateurs `uid`, `tempo`, `sync`, `scope`, *beatPos*, `expEvt` et `labels` ; $\tau.x$ est l'application de $\tau$ à $x$ ; $[x_1 \to e_1, \ldots x_i \to e_i]$ définit le contexte temporel $\tau$ tel que $\tau.x_i = e_i$ et qui est indéfini pour tous les autres identificateurs ; la composition $\tau_1\tau_2$ $(\tau_1\tau_2).x) = \tau_2.x$ si $\tau_2.x$ est défini et $\tau_1.x$ sinon ; $\tau + d$ note l'avancement du temps dans le contexte temporel : $\tau + d = \tau[beatPos \to \tau.beatPos + d]$ |
| $p \in \mathcal{P} = \mathcal{A} \star \mathcal{D}$ | les programmes temporisés (p. 105) |
| $p_\tau \in \mathcal{P}_\Upsilon = \mathcal{P} \otimes \Upsilon$ | les coroutines : un programme temporisé $p$ dans un contexte temporel $\tau$. Les coroutines sont ordonnées temporellement par $<_\rho$. (p. 105) |
| $\mathbf{P} \in \mathcal{P}_\Upsilon, \mathbf{R} \in \mathcal{R}_\Upsilon$ | les sous-ensembles de coroutines et les sous-ensembles de réaction (p. 107) |
| $eval : \mathcal{T} \to \mathcal{E} \to \mathcal{V}al$ | évaluation d'une expression (p. 114) |
| $next : \mathcal{T} \to \mathcal{P}_\Upsilon \to \mathcal{A} \times \mathcal{P} \times \Upsilon \times \mathcal{P}_\Upsilon$ | sélection de la prochaine coroutine à évaluer en déstructurant la coroutine en sa première action, la continuation, son environnement temporel et le reste des coroutines (p. 114) |
| $\Delta_{\rho,\tau}(d)$ | traduction d'une durée relative à un contexte temporel en une durée en seconde (p. 115) |
| $inBeats : \mathcal{T} \to \Upsilon \to \mathbb{Q} \to \mathbb{Q}$ | traduction d'une durée en seconde en une durée relative à un contexte temporel (p. 116) |
| $evalFirstDelay_\rho(p)$ | évaluation du délai de tête d'un programme (p. 116) |
| $erase_\ell(\mathbf{P})$ | suppression des actions portant un label donné (p. 116) |
| $\mathbf{Eval}[\![ \mathbf{P}, \mathbf{R}, \overline{\mathbf{R}} ]\!]\rho$ | fonction sémantique qui calcule la trace des coroutines (les équations définissant **Eval** par cas sont détaillées à partir de la page 119) |

FIGURE 31: Notations utilisées dans ce chapitre.

## 7.6 ÉTAT, TRACE ET SÉMANTIQUE D'UN PROGRAMME

La sémantique que nous voulons développer ici est *interactive* [Mos90] : les entrées d'un programme *Antescofo* sont fournies progressivement dans le temps, et le programme réagit continûment à ces entrées, sans attendre les entrées ultérieures. De la même manière, les sorties d'un programme *Antescofo* sont produites au fur et à mesure de l'exécution, sans attendre que l'ensemble des entrées ait été fourni. De plus, même si le programme se termine anormalement à un certain moment, des sorties ont pu être produites auparavant. Ce comportement contraste avec une vision « fonctionnelle » où un programme correspond à une fonction qui prend toute les données d'entrée « à la fois » pour les transformer en sortie.

Informellement, on peut essayer d'exprimer le caractère interactif de la sémantique de la manière suivante. Si une entréee $I$ donnée sous la forme d'une trace temporisée peut se découper en deux sous-traces $I = I' \bullet I''$, telles que $I'$ se termine en une date $t$, et si $S = S' \bullet S''$ est un programme tel que l'évaluation de $S'$ se termine en $t$, alors, on veut vérifier une propriété similaire à :

$$\text{trace}(S, I) = \text{trace}(S', I') \bullet \text{trace}(S'', I'')$$

qui exprime une contrainte de causalité (la trace jusqu'à $t$ ne dépend pas d'événement ultérieur à $t$) et à une propriété d'interactivité (même si l'évaluation de $S''$ se passe mal, la trace de sortie comporte le résultat de l'exécution de $S'$, car $\bullet$ n'est pas une opération stricte).

Cependant, l'équation précédente exprime une propriété bien plus forte : un calcul qui satisfait à cette condition est un *calcul sans état* puisqu'il ne permet pas qu'une sortie ultérieure dépende d'entrées antérieures. Pour modéliser une condition de causalité et une propriété d'interactivité plus large, il faut expliciter l'état d'un programme *Antescofo* en cours d'évaluation.

*État d'un programme*

Cet état sera représenté ici par un quadruple

$$(\mathbf{P}, \mathbf{R}, \overline{\mathbf{R}}, \rho)$$

où $\mathbf{P} \in \mathcal{P}_\Upsilon$ est un ensemble bien formé de coroutines en cours d'évaluation, $\mathbf{R} \in \mathcal{R}_\Upsilon$ est un ensemble fini de réactions actives, $\overline{\mathbf{R}} \subseteq \mathbf{R}$ est un sous-ensemble des réactions actives qui sont inhibées et $\rho \in \mathcal{T}$ est un environnement d'évaluation qui enregistre aussi les sorties du calcul.

L'évaluation d'un programme *Antescofo* $S$ est définie par la fonction

$$\mathbf{Eval} : \mathcal{P}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{T} \longrightarrow \mathcal{P}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{T}$$

où $\mathbf{Eval}[\![ \mathbf{P}, \mathbf{R}, \overline{\mathbf{R}} ]\!] \rho$ transforme l'état du programme en calculant l'application des réactions de $\mathbf{R} \backslash \overline{\mathbf{R}}$ aux coroutines de $\mathbf{P}$ dans l'environnement $\rho$. Cette fonction récursive est définie par quatre cas donnés

par $\text{next}_\rho(\mathbf{P})$. La récursion se fait sur le premier argument et le cas de base correspond à l'évaluation d'un ensemble vide de coroutines.

*Trace d'un programme*

Dans ce contexte, la trace de l'exécution d'une partition augmentée S en présence des entrées I est définie par :

$$\text{trace}(S, I) = \rho$$

avec

$$(\mathbf{P}, \mathbf{R}, \overline{\mathbf{R}}, \rho) = \mathbf{Eval}[\![\{I_{\tau_J}\}, \{S\}, \varnothing]\!]\ ((\texttt{\$NOW := 0}) \bullet (\texttt{\$RNOW := 0}))$$

Autrement dit, la trace de l'exécution d'un programme abstrait *Antescofo* est l'environnement créé par son évaluation. Et les entrées d'un programme sont vues comme une coroutine particulière qui « génère » les événements musicaux.

*Sémantique d'un programme*

L'objet mathématique $[\![S]\!] : \mathcal{T} \to \mathcal{T}$ que l'on associe à une partition augmentée S est la fonction qui produit la trace de l'exécution de S à partir d'une entrée :

$$[\![S]\!] = \lambda I.\text{trace}(S, I)$$

## 7.7 ÉQUATIONS SÉMANTIQUES DE LA FONCTION **Eval**

Il nous faut donc définir précisément la fonction **Eval**. Nous allons le faire à travers les équations données ci-après. Ces équations correspondent à la spécification de l'évolution de l'état du programme :

- l'évaluation d'une mise à jour dans une coroutine de **P** peut déclencher des réactions de $\mathbf{R}\backslash\overline{\mathbf{R}}$ ce qui va ajouter le corps de ces réactions dans l'ensemble **P** des coroutines en cours d'exécution et les inscrire dans l'ensemble des réactions inhibées $\overline{\mathbf{R}}$ ;
- une commande de terminaison supprime des coroutines dans **P** et des réactions de **R** ;
- un délai conduit à soustraire une certaine durée de toutes les coroutines en attente dans **P**.

Ces différentes causes d'évolution d'un état sont représentées à la figure 32 et se traduisent par les équations ci-dessous où les ensembles **P** et **R** sont supposés bien formés.

La fonction sémantique **Eval** qui associe une trace d'exécution à partir de l'état d'un programme est défini à travers 4 cas :

**(c1)** Le *cas de base* spécifie la trace d'un ensemble vide de coroutines.

**(c2)** Le *passage du temps* gère l'expiration d'un délai.

**(c3)** L'*exécution d'une action* définit l'effet d'une action sur l'état.

**(c4)** La *gestion des retards* prend en charge le comportement de l'évaluation en cas d'événement musical manqué.

Ces cas se caractérisent avec la fonction next qui retourne la prochaine action à effectuer ou bien le délai à attendre dans un ensemble de coroutines.

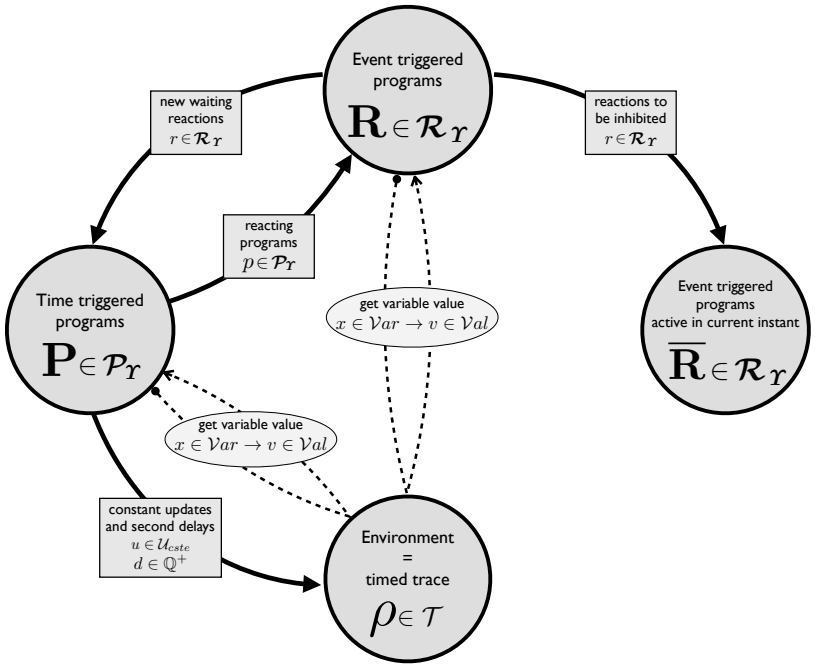



FIGURE 32: Diagramme présentant les différentes transitions affectant l'état d'un programme $(\mathbf{P}, \mathbf{R}, \overline{\mathbf{R}}, \rho)$, argument de la fonction $\mathbf{Eval}[\![\ ]\!]$.

**(C1) Cas de base :**

$$\text{next}_\rho(\mathbf{P}) = \bot$$

La fonction next n'est pas définie dans deux cas : s'il n'y a pas de coroutines, i. e. $\mathbf{P} = \varnothing$, ou bien si $\mathbf{P}$ n'est pas bien formé. On définit donc :

$$\mathbf{Eval}[\![ \varnothing, \mathbf{R}, \overline{\mathbf{R}} ]\!]\rho = (\varnothing, \mathbf{R}, \overline{\mathbf{R}}, \rho) \quad (2)$$

$$\mathbf{Eval}[\![ \mathbf{P}, \mathbf{R}, \overline{\mathbf{R}} ]\!]\rho = (\mathbf{P}, \mathbf{R}, \overline{\mathbf{R}}, \rho \bullet \bot) \quad \text{si } \mathbf{P} \text{ non bien formé} \quad (3)$$

L'équation (2) correspond à l'arrêt d'un programme car il n'y a plus de calcul à effectuer. Les entrées étant représentées par une coroutine, cela implique que les entrées sont finies. L'équation (3) correspond à l'arrêt d'un programme sur une erreur : l'ensemble de coroutines $\mathbf{P}$ à évaluer n'est pas bien formé, i. e., il y a plusieurs coroutines qui doivent évaluer une action à la même date et qui ont la même priorité (identifiant). Pour éviter un comportement non-déterministe, le calcul se termine en erreur et la trace résultante est partielle.

**(C2) Passage du temps :**

$$\text{next}_\rho(\mathbf{P}) = (d_{min}, p, \tau, \mathbf{P}')$$
avec isUndef($\tau$.expEvt) ou bien $\tau$.expEvt $\geqslant \rho($`$POS`$)

Dans ce cas, toutes les coroutines de **P** sont en attente de l'expiration d'un délai et il n'y a pas d'action à exécuter. Ce cas correspond donc au passage du temps.

$$\mathbf{Eval}[\![\, \mathbf{P}, \mathbf{R}, \overline{\mathbf{R}} \,]\!]\,\rho = \mathbf{Eval}[\![\, \mathbf{Q}, \mathbf{R}, \varnothing \,]\!]\,\rho \bullet d_{sec} \bullet (\texttt{\$NOW} \mathrel{+}= d_{sec}) \bullet (\texttt{\$RNOW} := t) \tag{4}$$

avec

$$\begin{aligned}
d_{sec} &= \Delta_{\rho,\tau}(d_{min}) \\
t &= \min\big(\texttt{\$RNOW} + d_{sec} * \rho(\texttt{\$RT_tempo}),\ \mathrm{sucScore}(\texttt{\$RNOW})\big) \\
\mathbf{Q} &= \{p_{\tau_1}\} \cup \big\{(d - d' \bullet q)_{\tau_3} \mid (d \bullet q)_{\tau_2} \in \mathbf{P}', d' = \mathrm{inBeats}_{\rho,\tau_2}(d_{sec})\big\} \\
\tau_1 &= \tau[\mathrm{beatPos} \to \tau.\mathrm{beatPos} + d_{min}, \\
&\qquad \tau.\mathrm{expEvt} \to (\tau.\mathrm{sync} = \mathrm{loose}\ ?\ \mathit{undef} : \tau.\mathrm{expEvt})] \\
\tau_3 &= \tau_2[\mathrm{beatPos} \to \tau.\mathrm{beatPos} + d', \\
&\qquad \tau.\mathrm{expEvt} \to (\tau.\mathrm{sync} = \mathrm{loose}\ ?\ \mathit{undef} : \tau.\mathrm{expEvt})]
\end{aligned}$$

la quantité $d_{sec}$ représente en seconde la durée $d_{min}$ qui vient de s'écouler. Cette durée est ajoutée à `$NOW`. La même durée exprimée en pulsation $t$ est ajoutée à `$RNOW` mais cette progression est bornée par la position du prochain événement attendu. Ces deux mises à jour se font dans l'environnement après l'écoulement de $d_{sec}$.

Le calcul de **Q** exprime l'effet du passage du temps sur les coroutines. Pour une coroutine $(d \bullet q)_{\tau_2}$ en attente de l'expiration d'un délai $d$ :

- l'attente d'une durée $d$ est réduite d'une durée $d'$ correspondant à la durée $d_{min}$ exprimée dans le contexte $\tau_2$ ;
- la position progresse de $d'$ dans le contexte temporel $\tau_3$.

On remarquera que le passage du temps annule l'inhibition des réactions : en effet, quand du temps métrique passe, on change d'instant. L'inhibition porte sur les réactions qui ont été activées dans l'instant, afin d'éviter les court-circuits temporels.

**(C3) Exécution d'une action :**

$$\begin{aligned}
&\mathrm{next}_\rho(\mathbf{P}) = (e, p, \tau, \mathbf{P}') \\
&\text{avec } \mathrm{isUndef}(\tau.\mathrm{expEvt}) \text{ ou bien } \tau.\mathrm{expEvt} \geqslant \rho(\texttt{\$POS})
\end{aligned}$$

Ce cas correspond au traitement d'une coroutine qui n'est pas en retard : soit le contexte temporel ne définit pas d'événement attendu ($\mathrm{isUndef}(\tau.\mathrm{expEvt})$) soit le prochain événement attendu par le contexte temporel n'est pas encore arrivé.

Dans ce cas, quand une action $e$ est prête à être exécutée et qu'elle est suivie d'une continuation $p$, l'évolution de l'état est spécifiée par :

$$\mathbf{Eval}[\![\, \mathbf{P}, \mathbf{R}, \overline{\mathbf{R}} \,]\!]\,\rho = \mathbf{Eval}[\![\, \mathbf{P}_e \cup \mathbf{P}_p, \mathbf{R}_e \cup \mathbf{R}_p, \overline{\mathbf{R}}_e \,]\!]\,\rho_e \tag{5}$$

où

$$(\mathbf{P}_e, \mathbf{R}_e, \overline{\mathbf{R}}_e, \rho_e) = \mathbf{Eval_e}[\![ e, \mathbf{P}', \mathbf{R}, \overline{\mathbf{R}} ]\!]\,\rho\,.$$

Les ensembles $\mathbf{P}_p$ et $\mathbf{R}_p$ sont utilisés pour gérer la synchronisation de la continuation $p$. Ils sont définis plus bas. La trace $\rho_e$ et les ensembles $\mathbf{P}_e, \mathbf{R}_e$ et $\overline{\mathbf{R}}_e$ correspondent à l'évolution de $\rho$, $\mathbf{P}$, $\mathbf{R}$ et $\overline{\mathbf{R}}$ suite à l'évaluation de $e$. Ils sont définis par la fonction auxiliaire :

$$\mathbf{Eval_e} : \mathcal{A} \times \mathcal{P}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{T} \longrightarrow \mathcal{P}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{T}$$

appliquée à l'action $e$, la continuation $p$, les réactions $\mathbf{R}$ et $\overline{\mathbf{R}}$, et la trace $\rho$. La définition de $\mathbf{Eval_e}$ se fait par cas sur la structure de l'action $e$ :

$$\mathbf{Eval_e}[\![ e, \mathbf{P}', \mathbf{R}, \overline{\mathbf{R}} ]\!]\,\rho = \begin{cases} \left(\{\text{evalFirstDelay}_\rho(q)_{\tau_2}\} \cup \mathbf{P}',\ \mathbf{R},\ \overline{\mathbf{R}},\ \rho\right) & \text{si } e = q_{\tau_1} \in \mathcal{P}_\Upsilon \quad (\textit{group}) \\ \left(\mathbf{P}',\ \{e_\tau\} \cup \mathbf{R},\ \overline{\mathbf{R}},\ \rho\right) & \text{si } e \in \mathcal{R} \quad (\textit{reaction}) \\ \left(\text{erase}_{\mathtt{id}}(\mathbf{P}'),\ \text{erase}_{\mathtt{id}}(\mathbf{R}),\ \text{erase}_{\mathtt{id}}(\overline{\mathbf{R}}),\ \rho\right) & \text{si } e = (\mathtt{abort\ id}) \quad (\textit{abort}) \\ \left(\mathbf{Q}_p \cup \mathbf{P}',\ \mathbf{R},\ \mathbf{Q}_r \cup \overline{\mathbf{R}},\ \rho'\right) & \text{si } e = (x := \text{exp}) \quad (\textit{update}) \end{cases}$$

avec :

$$\begin{aligned} \tau_2 &= \tau\tau_1[\mathtt{uid} \to \textsc{Fresh}, \\ &\qquad \tau.\text{expEvt} \to (\tau.\text{sync} = \text{loose}\ ?\ \textit{undef} : \tau.\text{beatPos})] \\ \rho' &= \rho \bullet (x := \text{eval}_\rho(\text{exp})) \\ \mathbf{Q}_r &= \{ r_{\tau_3} \mid r_{\tau_3} \in \mathbf{R} \backslash \overline{\mathbf{R}}, r = (\text{exp} \xrightarrow{V} q) \wedge (x \in V) \wedge (\text{eval}_{\rho'}(\text{exp}) = \text{true}) \} \\ \mathbf{Q}_p &= \left\{ \text{evalFirstDelay}_{\rho'}(q)_{\tau\tau_3[\mathtt{uid} \to \textsc{Fresh}]} \mid (\text{exp} \xrightarrow{V} q)_{\tau_3} \in \mathbf{Q}_r \right\} \end{aligned}$$

Ces équations s'interprètent de la manière suivante :

- Si $e$ est une coroutine $q_{\tau_1}$, on va insérer une nouvelle coroutine dans $\mathbf{P}$. Le code de cette coroutine est celui de $q$, à l'exception d'un éventuel premier délai qui sera évalué (les délais sont évalués au moment où on lance l'action qui précède). Le contexte temporel de cette nouvelle coroutine est $\tau_2$ qui complète les information de $\tau_1$ avec celle du contexte courant $\tau$ et un nouvel identifiant. Le lancement de la nouvelle coroutine n'est pas observable depuis l'extérieur et la trace reste inchangée.
- Si $e$ est une réaction cette réaction est rajoutée dans l'ensemble des réactions actives $\mathbf{R}$, avec un contexte temporel qui correspond à l'environnement courant $\tau$. Cela implique que lorsque que la réaction se déclenche, le contexte temporel des actions déclenchées sera celui de la définition de la réaction : on a là une liaison statique.
- Si $e$ est une commande de terminaison, on efface de $\mathbf{P}$, $\mathbf{R}$ et $\overline{\mathbf{R}}$ toutes les entités qui sont étiquetées par le bon label.

– Si $e$ est une affectation, alors l'ensemble des coroutines en cours d'exécution est augmenté du produit des réactions $\mathbf{Q}_R$ déclenchées. Ces réactions sont les éléments $r_{\tau_3}$ de $\mathbf{R}$ qui surveillent la variable affectée et dont la garde s'évalue à vrai. Ces réactions sont inhibées dans la suite. Le produit d'une réaction $r_{\tau_3}$ est une coroutine $q_{\tau_3}$ constituée de $r$ et d'un contexte temporel qui complète $\tau_3$ par les informations du contexte courant et par un nouvel identifiant.

Nous définissons à présent les ensembles $\mathbf{P}_P$ et $\mathbf{R}_p$ qui sont les ajustements nécessaires au traitement de la continuation.

*Gestion de la continuation*

Les ensembles $\mathbf{P}_p$ and $\mathbf{R}_p$ sont définis par les cas qui déterminent le devenir de la continuation $p$ :

1. Le premier cas correspond à un suicide : l'action $e$ est une commande `abort` qui amène à la terminaison de la coroutine dont est issue $e$. Dans ce cas, il n'y a pas de continuation $p$ à garder dans l'ensemble des coroutines actives.
2. Le second cas correspond à la gestion des continuations $p$ qui débutent par un délai. Ce délai doit être évalué dans l'environnement et le contexte temporel courant pour connaître la durée à attendre. Remarquons qu'on évalue bien la durée d'un délai lors du lancement de l'action qui le précède. Il faut traiter deux cas suivant que le délai est `@loose` ou `@tight` :
   a) Si le délai est `@loose`, il n'y a rien de plus à faire : la continuation $p$ est rajoutée dans les coroutines actives avec son délai de tête évalué mais avec le même contexte temporel.
   b) Si le délai est `@tight` ce délai doit être non seulement évalué mais aussi ajusté pour démarrer non pas dans l'instant, mais avec l'occurrence de l'événement musical précédent le plus proche dans la partition. Cela amène à considérer deux sous-cas :
      i. L'événement précédent le plus proche a déjà eu lieu et donc l'écoulement du délai qui débute la continuation a déjà commencé. Le contexte temporel de cette continuation est mis à jour afin que $\mathit{expEvt}$ réfère à la position de l'événement déclencheur. Ainsi, si un événement ultérieur arrive avant l'expiration du délais, la continuation sera considérée comme en retard. Le traitement des coroutines en retard est vu au cas (C4).
      ii. L'événement précédent le plus proche n'est pas encore arrivé : dans ce cas, on crée une réaction qui attend cet événement pour déclencher la continuation après un $d_{rem}$ courant à partir de la détection de cet événement.

Le contexte temporel de cette continuation doit être calculé pour rendre compte de la position correcte au réveil.

La distinction entre les deux sous cas est déterminée par le booléen *evtBefore* dans les équations ci-dessous : il est vrai dans le premier cas et faux dans le second.

3. Le dernier cas correspond à une continuation qui débute par une action qui n'est pas une terminaison. Il n'y a alors rien de spécial à faire : la continuation sera rajoutée dans les coroutines actives.

Cette analyse se formalise à travers les définitions suivantes :

$$\mathbf{P}_p = \begin{cases} \varnothing & \text{si } e = (\mathtt{abort}\ \ell) \wedge \ell \in \tau.\mathtt{labels} & \textit{(cas 1)} \\ \{(d' \bullet p')_\tau\} & \text{si } p = d \bullet p' \wedge \tau.\mathtt{sync} = \mathtt{loose} & \textit{(cas 2a)} \\ \varnothing & \text{si } p = d \bullet p' \wedge \tau.\mathtt{sync} = \mathtt{tight} \wedge \textit{evtBefore} & \textit{(cas 2(b)i)} \\ \{(d_t \bullet p')_{\tau_p}\} & \text{si } p = d \bullet p' \wedge \tau.\mathtt{sync} = \mathtt{tight} \wedge \neg \textit{evtBefore} & \textit{(cas 2(b)ii)} \\ \{p_\tau\} & \text{sinon} & \textit{(cas 3)} \end{cases}$$

$$\mathbf{R}_p = \begin{cases} \left\{(\textit{afterTime} \xrightarrow{\{\$\mathtt{POS}\}} (\$\mathtt{once} := 1) \bullet d_{rem} \bullet p')_{\tau_r}\right\} & \\ \qquad \text{si } p = d \bullet p' \wedge \tau.\mathtt{sync} = \mathtt{tight} \wedge \textit{evtBefore} & \textit{(cas 2(b)i)} \\ \varnothing & \\ \qquad \text{sinon} & \textit{(cas 1)}\ \textit{(cas 2a)}\ \textit{(cas 3)}\ \textit{(cas 2(b)ii)} \end{cases}$$

avec

$$\begin{aligned} d' &= \mathrm{eval}_{\rho_e}(d) \\ pos &= \tau.\mathtt{beatPos} + d' \\ d_t &= pos - \rho_e(\$\mathtt{RNOW}) \\ \textit{precEvent} &= \mathrm{precScore}(pos) \\ \textit{evtBefore} &= \textit{precEvent} > \tau.\mathtt{beatPos} \wedge \textit{precEvent} > \rho_e(\$\mathtt{RNOW}) \\ \textit{afterTime} &= \rho_e(\$\mathtt{POS}) \geqslant \textit{precEvent} \wedge \mathrm{isUndef}(\$\mathtt{once}) \\ d_{rem} &= pos - \textit{precEvent} \\ \tau_p &= \tau[\mathtt{expEvt} \to \textit{precEvent}] \\ \tau_r &= \tau[\mathtt{beatPos} \to \tau.\mathtt{beatPos} + d' - d_{rem}, \tau.\mathtt{expEvt} \to \textit{precEvent}] \end{aligned}$$

La variable pos représente la position dans la partition après l'expiration du délai d : l'expression $\Delta_{\rho_e,\tau}(d)$ calcule la durée en seconde du délai d exprimé dans le contexte temporel $\tau$. Cette durée en seconde est convertie en pulsation dans le contexte $\tau_0$ du musicien. La

variable *precEvent* est la position dans la partition de l'événement musical qui précède la position pos. Le booléen *evtBefore* est vrai si l'événement *precEvent* n'est pas encore arrivé. La valeur $d_{rem}$ représente la durée qui reste à attendre après l'occurrence *precEvent* exprimée dans le contexte courant afin de réaliser une attente de d. L'expression *afterTime* est une expression *Antescofo* dont la valeur booléenne est utilisée dans la condition de la réaction qui attend l'événement à partir duquel le délai tight sera décompté (ou un événement ultérieur).

**(C4) Gestion des retards :**

$$\begin{aligned}&\text{next}_\rho(\mathbf{P}) = (x, p, \tau, \mathbf{P}')\\&\text{avec } \tau.expEvt < \rho(\$POS)\end{aligned}$$

Ce dernier cas correspond à la gestion des coroutines en retard. Une coroutine est en retard si le paramètre expEvt de son contexte temporel est défini et inférieur à la dernière position détectée par la machine d'écoute. Une coroutine loose ne peut être en retard qu'au moment de sa création. Si une coroutine en retard débute :

- par une action `@local`, cette action n'est pas exécutée et on passe à la suite ;
- par une action `@global` : cette action est exécutée normalement, comme au cas précédent, et on passe à la suite ;
- par un délai : la durée correspondant à ce délai est utilisée pour mettre à jour les paramètres beatPos et expEvt du contexte temporel. Si ce délai est assez grand pour absorber le retard, on gère la continuation comme dans le cas précédent.

Ces considérations conduisent aux équations suivantes :

$$\mathbf{Eval}[\![\, \mathbf{P}, \mathbf{R}, \overline{\mathbf{R}} \,]\!]\rho = \begin{cases} \mathbf{Eval}[\![\, \mathbf{P}_e \cup \{p_\tau\}, \mathbf{R}_e, \overline{\mathbf{R}}_e \,]\!]\rho_e & \text{si } x \in \mathcal{A} \wedge \tau.scope = global \\ \mathbf{Eval}[\![\, \mathbf{P}' \cup \{p_\tau\}, \mathbf{R}, \overline{\mathbf{R}} \,]\!]\rho & \text{si } x \in \mathcal{A} \wedge \tau.scope = local \\ \mathbf{Eval}[\![\, \mathbf{P}' \cup \mathbf{P}'_p, \mathbf{R}' \cup \mathbf{R}'_p, \overline{\mathbf{R}} \,]\!]\rho & \text{si } x \in \mathcal{D} \end{cases} \tag{6}$$

avec $\mathbf{P}_e$, $\mathbf{R}_e$, $\overline{\mathbf{R}}_e$ et la trace $\rho_e$ définis comme dans le cas précédent. Les ensembles $\mathbf{P}'_p$ et $\mathbf{R}'_p$ sont définis quand $x$ est un délai :

$$\mathbf{P}'_p = \begin{cases} \{p_{\tau_{P1}}\} & \text{si } \mathtt{stayLate} \\ \varnothing & \text{si } \neg\mathtt{stayLate} \wedge \tau.\mathtt{sync} = \mathtt{tight} \wedge \mathit{evtBefore} \\ \{(d' \bullet p)_{\tau_{P2}+\mathtt{retard}}\} & \text{si } \neg\mathtt{stayLate} \wedge \tau.\mathtt{sync} = \mathtt{tight} \wedge \neg\mathit{evtBefore} \\ \{(d' \bullet p)_{\tau_{P3}+\mathtt{retard}}\} & \text{si } \neg\mathtt{stayLate} \wedge \tau.\mathtt{sync} \neq \mathtt{tight} \end{cases}$$

$$\mathbf{R}'_p = \begin{cases} \left\{(\mathit{afterTime} \xrightarrow{\{\$\mathtt{POS}\}} (\$\mathtt{once} := 1) \bullet d_{rem} \bullet p)_{\tau_R+(d'-d_{rem})}\right\} & \\ \qquad \text{si } \neg\mathtt{stayLate} \wedge \tau.\mathtt{sync} = \mathtt{tight} \wedge \mathit{evtBefore} & \\ \varnothing & \\ \qquad \text{elsewhere} & \end{cases}$$

avec

$$\begin{aligned} pos_e &= \rho(\$\mathtt{POS}) \\ d' &= \mathrm{eval}_\rho(x) \\ \mathtt{retard} &= pos_e - \tau.\mathtt{expEvt} \\ \mathtt{stayLate} &= d' < \mathtt{retard} \\ pos &= \tau.\mathtt{expEvt} + d' \\ \mathit{precEvent} &= \mathrm{precScore}(pos) \\ \mathit{evtBefore} &= \mathit{precEvent} > \tau.\mathtt{beatPos} \wedge \mathit{precEvent} > \rho_e(\$\mathtt{RNOW}) \\ d_{rem} &= pos - \mathrm{precEvt} \\ \mathit{afterTime} &= \rho(\$\mathtt{POS}) \geqslant \mathit{precEvent} \wedge \mathrm{isUndef}(\$\mathtt{once}) \\ \tau_{P1} &= \tau[\mathtt{beatPos} \to \tau.\mathtt{beatPos} + d', \\ &\qquad \tau.\mathtt{expEvt} \to \tau.\mathtt{expEvt} + d'] \\ \tau_{P2} &= \tau[\tau.\mathtt{expEvt} \to \mathit{precEvent}] \\ \tau_{P3} &= \tau[\tau.\mathtt{expEvt} \to \mathit{undef}] \\ \tau_R &= \tau[\tau.\mathtt{expEvt} \to \mathit{precEvent}] \end{aligned}$$

La variable $pos_e$ désigne la position du dernier événement détecté (le plus récent) ; $d'$ est la valeur du délai d exprimé dans le contexte temporel courant ; retard exprime le retard à rattraper.

Le booléen stayLate est vrai si le délai à attendre n'est pas assez grand pour absorber le retard.

Les variables pos, *precEvent*, *evtBefore*, $d_{rem}$ et *afterTime* correspondent aux quantités définies pour le cas précédent (C3).

Le contexte temporel $\tau_{P1}$ correspond à la mise à jour de $\tau$ avec une nouvelle position pour l'événement attendu expEvt et pour le paramètre *beatPos*. Puisque la séquence courante a avancé d'un délai

d, la valeur de $expEvt$ est incrémentée de $d'$ pour qu'à la prochaine étape, on puisse connaître le retard restant à rattraper.

Les contextes temporels $\tau_{P2}$ et $\tau_R$ sont mis à jour comme dans la gestion de la continuation au cas précédent.

Le paramètre $expEvt$ dans $\tau_{P3}$ est mis à *undef* car ce contexte est utilisé quand la synchronisation est loose, et une fois lancée, la coroutine n'est plus en retard.

## 7.8 TEMPO DÉFINI IMPLICITEMENT PAR UNE STRATÉGIE ANTICIPATIVE

La synchronisation d'un contexte temporel avec un autre au moyen d'une stratégie anticipative définit implicitement le tempo du premier. Dans cette section nous donnons explicitement l'expression de cette fonction, qui est utilisée dans la traduction entre une durée exprimée en pulsation et une durée exprimée en seconde, cf. sections 7.5.4 et 7.5.5.

### 7.8.1 *Dynamic target*

Soit $\tau$ un contexte temporel synchronisé par une stratégie `@target` $[w]$ avec un processus caractérisé par une position $pos$ et un tempo $tempo$.

Le processus $T_{ref}$ peut être le musicien : c'est alors la machine d'écoute qui fournit $pos$ et $tempo$ comme valeur des variables `$BEAT_POS` et `$RT_TEMPO`. Mais cela peut être aussi n'importe quel processus externe qui fournit ces deux quantités [5]. L'utilisation dans la suite des variables $pos$ et $tempo$ a pour objectif de s'abstraire du processus $T_{ref}$ avec lequel on veut se synchroniser.

La stratégie de synchronisation anticipative dynamique se définit par la propriété suivante :

> si au temps $t$ la position $\tau.beatPos$ diffère de $pos$ d'un retard de 1 pulsation, alors au temps $t + w$ la position de $\tau$ et $T_{ref}$ coïncident, ainsi que leur tempo (en supposant que le tempo de $T_{ref}$ reste constant et qu'aucun événement de $T_{ref}$ n'amène à réviser la position).

Le bénéfice apporté par cette caractérisation est l'*invariance* de la stratégie par rapport au temps physique : le tempo exprimé ne dépend que du retard de position entre $\tau$ et $T_{ref}$ et du tempo courant de $T_{ref}$. Il ne dépend pas des actions à synchroniser ou du temps qui passe.

Le problème est de passer de cette caractérisation implicite à une formule explicite du tempo. Pour y arriver, nous postulerons que la

5. En *Antescofo* on peut par exemple calculer le tempo associé à la mise à jour d'une variable, en utilisant l'algorithme de E. Large [LJ99]. Et on peut donc se synchroniser sur les mises à jours d'une variable comme on se synchronise sur le musicien.

courbe de tempo qui satisfait cette propriété est une fonction $f$, linéaire par morceaux, sur deux deux intervalles. Le premier intervalle, $g$, qui part de l'instant courant, correspond au rattrapage de la position et du tempo de la cible $T_{ref}$ par une variation linéaire. Sur l'intervalle qui suit, le tempo de $\tau$ reste constant et égal à $tempo$. Une fois la fonction $g$ exprimée comme fonction du temps (en seconde), l'intégrale $G$ de $g$ définit les positions futures de $\tau$ tant que $\tau.beatPos \neq pos$ (toujours en supposant que $tempo$ reste constant). Cette approche appelle deux remarques :

- L'expression explicite de $f$ comme une fonction du temps physique n'est pas immédiate : la date $t'$ à laquelle $g(t') = tempo$ dépend de la la différence entre $\tau.pos$ et $pos$ au temps $t$.
- Le choix d'une fonction linéaire pour atteindre le tempo cible impose potentiellement une discontinuité dans le tempo au temps initial. Toutefois la position elle reste continue : il n'y a pas de saut. Une alternative serait de choisir un polynôme de degré deux pour contraindre le tempo à l'instant de départ aussi bien qu'à l'instant de rattrapage. Malheureusement, dans ce cas, la courbe de tempo ne sera plus nécessairement monotone, ce qui est une propriété naturelle, on ne peut pas borner l'accroissement du tempo (il existe des configuration où le tempo accélère puis ralenti afin de contraindre la cubique) et on ne peut plus interpréter naturellement le facteur $w$ comme le temps de rattrapage d'une erreur unitaire (ce qui est intuitif).

La situation est illustrée à la figure 33 où l'on suppose que l'on a une différence positive $diff = \tau.beatPos - position$ à rattraper (i. e., la position de $\tau$ est plus grande que $pos$ et $\tau$ est donc en avance).

Dans cette figure, l'axe horizontale $Ox$ correspond au temps physique en seconde et l'axe vertical $Oy$ correspond à la position en pulsation. *Cependant*, l'origine de ce repère est fixé par translation de manière à ce que la position de $T_{ref}$ en $0$ soit $0$ et $G(0) = 1$ (la position de $\tau$ au temps $0$ est 1). Ces translations ont pour avantage de rendre l'expression directe de $\tau.beatPos$ plus simple. En effet, l'expression de $G$ dans ce système de coordonnées est :

$$G(x) = \frac{x^2}{w^2} + (tempo - \frac{2}{w})x + 1$$

car on peut vérifier immédiatement que $G$ est l'intégrale d'une fonction linaire de $x$, $G(0) = 1$ et $G(w) = tempo * w$, ce qui est la position atteinte par $T_{ref}$ au bout de $w$ secondes.

Pour calculer le nombre de secondes $dsec$ correspondant à un délai $d$ en pulsation démarrant à l'instant courant (de date `$NOW` en temps physique), nous pouvons exprimer la translation de l'origine $Ox$ et utiliser la formule générale donnée à la section 7.5.4. Cependant, il est possible de calculer directement cette quantité à partir du diagramme de la figure 33 :

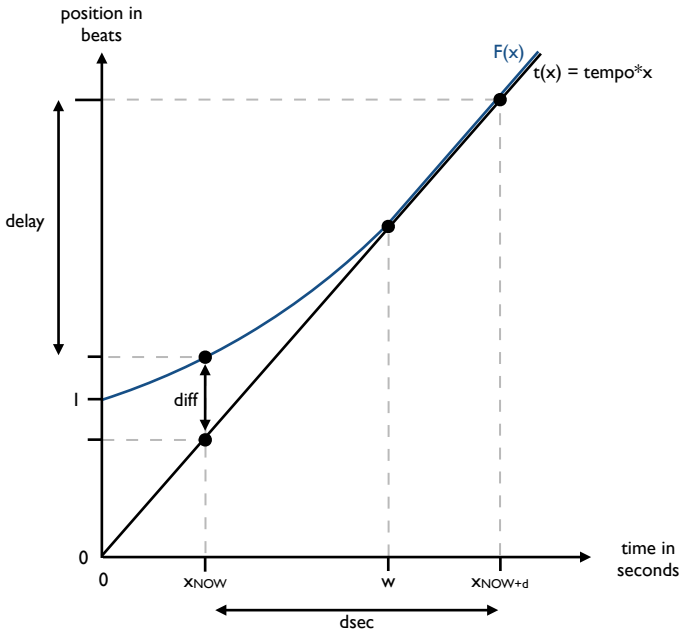



FIGURE 33: Méthode utilisée pour calculer la durée dsec en secondes correspondant au delai d en pulsation avec une cible dynamique $[w]$ et une différence initiale en position de $diff = \tau.beatPos - position$. La fonction F represente la position de $\tau$ comme fonction du temps x. Cette fonction est constitué de deux parties : la première, G où $\tau.tempo$ varie linéairement jusqu'à devenir égal à tempo. A partir de cette date, F evolue comme pos avec un tempo $\tau.tempo = tempo$ (une constante). La fonction G est la partie de parabole qui va de $x = 0$ à $x = w$. La localisation de la date $x_{NOW}$ correspondant à l'instant courant dans le repère Ox est effectuée en recherchant le point de Ox qui réalise une différence de *diff* entre la position de $\tau$ et pos.

– Nous calculons tout d'abord la date $x_{NOW}$ sur Ox tel que $G(x_{NOW}) - tempo * x_{NOW} = diff$. L'abscisse $x_{NOW}$ correspond à une différence de diff entre les positions et correspond donc à l'instant courant. De l'expression de G donnée plus haut, on dérive immédiatement que

$$x_{NOW} = w * \sqrt{diff}$$

– On calcule ensuite $x_{NOW+d}$ comme antécédent de $F(x_{NOW}) + d$.
– Et finalement on calcule $dsec = x_{NOW+d} - x_{NOW}$.

Le calcul précédent est valide si l'expression f du tempo est positive en $x_{NOW}$ (ce qui suffit à assurer que le tempo restera positif dans le futur). Si ce n'est pas le cas, alors il n'existe pas de croissance ou décroissance linéaire g du tempo qui permette de satisfaire la contrainte donnée plus haut. Dans ce cas, on suppose que le tempo est défini sur trois intervalles au lieu de deux : sur le premier intervalle, le tempo est fixé à 0, i. e. la position de $\tau$ est figée, jusqu'à ce que la différence *diff* soit suffisamment petite pour admettre une solution. Le tempo

évolue ensuite linéairement puis reste constant comme expliqué ci-dessus.

Si la différence est négative, la fonction G se définit ainsi (par le même raisonnement que précédemment) :

$$G(x) = -\frac{x^2}{w^2} + (tempo + \frac{2}{w})x - 1$$

Dans ce cas $g = G'$ est toujours positive à partir de $x_{NOW}$ et jusqu'à $w$ et l'expression de dsec est dérivée comme ci-dessus.

### 7.8.2 *Static target*

La fonction de tempo correspondant à une stratégie de synchronisation anticipative statique est calculée comme pour les stratégies dynamiques. La seule différence est que $w$ n'est pas fixé à l'avance mais est calculé comme la différence de position à la prochaine cible et que cette différence est réévaluée à chaque occurrence d'un événement (i. e., à chaque notification d'un changement de pos).

## 7.9 REMARQUES SUR LA SÉMANTIQUE

**LA FONCTION Eval⟦ ⟧ EST BIEN DÉFINIE.** On vérifie immédiatement que les équations (2–6) s'appliquent à des cas disjoints. Ces cinq équations définissent ensemble une fonction

$$\begin{aligned} \varphi \quad &: \quad D \to D \\ &\quad \text{avec } D = \mathcal{P}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{T} \longrightarrow \mathcal{P}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{R}_\Upsilon \times \mathcal{T} \end{aligned}$$

de sorte que la fonction sémantique est définie comme un point fixe de $\varphi$ sur D :

$$\mathbf{Eval} = \varphi(\mathbf{Eval})$$

D est un domaine obtenu comme produit cartésien des domaines $\mathcal{P}_\Upsilon$, $\mathcal{R}_\Upsilon$ et $\mathcal{T}$ décrits précédemment. La fonction $\varphi$ est continue sur ce domaine D, comme composition de fonctions continues. On sait alors qu'un plus petit point fixe existe [Mos90] et c'est ce plus petit point fixe qui est la fonction **Eval**.

**IMPLÉMENTATION DIRECTE DES ÉQUATIONS SÉMANTIQUES.** Les équations sémantiques (2–6) se traduisent directement en des fonctions ML définies par filtrage, ce qui donne une version exécutable de la sémantique, présentée en annexe. La fonction $\lambda I.\lambda S.\llbracket S \rrbracket I$ réalise un interprète du langage.

Dans cette implémentation, la construction des séquences temporisées est réalisée par un type algébrique : l'interprète n'est donc pas temps réel, mais peut servir d'oracle, par exemple pour le test.

L'ÉVALUATION D'UN PROGRAMME EST BIEN DÉFINIE SI L'ÉVALUATION DES EXPRESSIONS EST BIEN DÉFINIE. L'évaluation d'un programme $S$ *se termine en erreur* sur l'entrée $I$ si $\text{trace}(S, I)$ est une séquence partielle, i. e., prend la forme $x_1 \bullet x_2 \bullet \cdots \bullet x_n \bullet \bot$.

En inspectant les équations (2–6), on vérifie qu'un programme $S$ se termine en erreur sur $I$ uniquement si la première composante $\mathbf{P}$ de l'état n'est pas bien formée, ou bien si une évaluation $\text{eval}_\rho(d)$ d'une expression $d$ retourne $\bot$.

Si $\mathbf{P}$ est un ensemble bien formé de coroutines, alors :
- $\mathbf{Q}$ dans l'équation (4),
- $\mathbf{P}_e \cup \mathbf{P}_p$ dans l'équation (5), et
- $\mathbf{P}_e \cup \{p_\tau\}$, $\mathbf{P}' \cup \{p_\tau\}$ et $\mathbf{P}' \cup \mathbf{P}_p$ dans l'équation (6)

sont bien formés car les contextes temporels qui apparaissent dans ces ensembles ont tous des identifiants différents, comme on peut le vérifier dans les définitions correspondantes : un identifiant « frais » est généré à chaque fois qu'une nouvelle coroutine est ajoutée à un de ces ensembles. S'ils étaient bien formée, ils le restent donc.

Autrement dit, si $\mathbf{Eval}[\![\ ]\!]$ est appelé avec un ensemble de cororutines bien formé, les appels successifs se font sur des arguments bien formés. La valeur initiale du premier argument de $\mathbf{Eval}[\![\ ]\!]$ est l'ensemble $\{I_{\tau_J}\}$ qui est bien formé si la trace est bien formée, ce qui est le cas par construction. La valeur initiale du second argument est aussi bien formée par construction (un ensemble de réactions représentant le programme $S$). Par induction, la fonction eval est toujours appelée avec des arguments bien formés.

Autrement dit, si l'évaluation de $S$ se termine en erreur, c'est qu'une expression s'évalue en $\bot$.

LE CALCUL DES DURÉES EST COHÉRENT. Les délais qui permettent au temps de s'écouler, sont comptabilisés dans des unités de temps différentes, qui correspondent à des tempos différents. Il est donc possible que l'instant courant soit référé de manière différente et incohérente dans les différents contextes temporels. Pour que les différentes comptabilités soient cohérentes, il faut que les fonctions $\text{inBeats}_{\rho,\tau}$ et $\Delta_{\rho,\tau}$ soient en quelque sorte « inverses l'une de l'autre ». Ainsi, le passage du temps traité au cas (C2) est cohérent, puisque tout écoulement d'un délai est d'abord traduit en temps physique avant d'être retraduit et soustrait. La fonction $\Delta_{\rho,\tau}$ prenant en argument des expressions *Antescofo*, il faut cependant exprimer cette propriété sur les fonctions associées portant les délais numériques, après évaluation des expressions.

Soit $\Delta : \mathbb{Q} \to \mathbb{Q}$ définit par :

$$\Delta^f(d) = d' \text{ avec } d = \int_o^{o+d'} |f(t)|\,dt$$

Alors $\Delta_{\rho,\tau}(exp) = \Delta^{f'}(d)$ avec $d = \max(0, eval_\rho(exp))$ et $f' = |eval_\rho(\tau.tempo)|$, cf. paragraphe 7.5.4. On définit de même :

$$inBeats^f(d) = \int_o^{o+d} f(t)\,dt$$

et on a alors : $inBeats_{\rho,\tau}(d) = inBeats^{|eval_\rho(\tau.tempo)|}(d)$. Et la propriété souhaitée s'exprime alors :

$$d = inBeats^f(\Delta^f(d))$$
$$d_{sec} = \Delta^f(inBeats^f(d_{sec}))$$

pour toute fonction $f$ positive et $d, d_{sec}$ positifs.

Supposer que les durées et les tempos à traduire sont positifs ou bien nuls, ne suffit pas. La fonction $\Delta^f$ est une fonction continue et croissante car $f$ est positive. Cette fonction passe donc par tous les points entre 0 et sa borne supérieure M sur $[0, +\infty[$. Il faut que cette fonction soit strictement croissante et que $d < M$ pour qu'il existe une solution unique à l'équation définissant $d_{sec}$ :

- La fonction $\Delta^f$ n'est pas strictement croissante si $f$ est nulle sur un intervalle non réduit à un point. Ce cas correspond à un *tempo nul*. Pour un tempo nul, il existe plusieurs $d_{sec}$ qui répondent à la contrainte. Il faut choisir le plus petit.
- Si $d > M$, cela veut dire que la progression du temps est bornée et donc que le tempo tend vers zéro suffisamment vite.

Ces considérations ne sont pas que théorique. Par exemple, pour le premier cas, un compositeur peut vouloir « geler » l'activité d'un processus en fixant son tempo à 0, pour le « dégeler » ensuite.

Pour le second cas, « la fin des temps » dans un certain contexte temporel, se produit avant une date finie du temps physique. C'est l'analogue d'un *comportement de Zenon* : il suffit d'imaginer une boucle périodique infini dans ce contexte, pour voir que cela implique d'effectuer une infinité de calcul avant une date finie. Remarquons qu'on peut exprimer directement un comportement de Zenon même avec un tempo fixe :

```
$p := 1
Loop $p { $p := $p / 2 }
```

est une boucle qui divise sa période par deux à chaque itération.

LES PROGRAMMES *antescofo* SONT CAUSAUX. Nous voulons montrer que la trace obtenue à un instant donné de l'exécution du programme, ne dépend pas des entrées futures. La manière dont nous avons présenté la sémantique devrait rendre cette propriété évidente. Mais deux difficultés techniques viennent compliquer l'établissement de cette propriété.

La première, est qu'il faut que les calculs des délais soit cohérent, afin qu'un changement d'unités ne permette pas de référer au futur. Ce point a été discuté au paragraphe précédent.

La seconde difficulté est que la séquence complète des entrées est fournie en argument d'un programme. Cette séquence représente la suite des entrées dans le temps, mais rien n'interdit au niveau du formalisme, d'utiliser cette séquence « en avant » du rang représentant l'instant courant.

On peut se convaincre en examinant la définition de la fonction **Eval** que ce n'est pas le cas. Par exemple dans le cas (C2) correspondant au passage du temps, l'évolution d'un programme n'utilise dans **P** que son plus petit élément, qui correspond au délai minimal en seconde. Il n'y a donc pas de référence en avant.

Une manière alternative de montrer le caractère causal de l'évaluation d'un programme *Antescofo* est de profiter que la séquence d'entrée se retrouve dans la trace d'exécution. On peut donc se donner un « marqueur » dans la séquence d'entrée, par exemple l'affectation unique d'une variable distinguée `$M`, et montrer que la trace du programme jusqu'au marqueur, ne dépend pas des entrées ultérieures :

$$\begin{aligned} \mathit{trace}(S,\ I \bullet (\texttt{\$M} := 0) \bullet I') = J \bullet (\texttt{\$M} := 0) \bullet I'' \Rightarrow \\ \forall K, \exists K',\ \mathit{trace}(S,\ I \bullet (\texttt{\$M} := 0) \bullet K) = J \bullet (\texttt{\$M} := 0) \bullet K' \end{aligned}$$

Nous ne montrerons pas formellement cette propriété.

# 8 IMPLÉMENTATION

Dans ce chapitre nous donnons quelques éléments sur le développement du système *Antescofo*. Ce développement est un effort collectif qui se partage entre la machine d'écoute (Arshia Cont, Philippe Cuvillier) et la machine réactive (Jean-Louis Giavitto et moi-même). J'ai pris en charge cette dernière, avec en particulier la réalisation complète de tout ce qui concerne la gestion du temps : ordonnanceur, stratégies de synchronisation, calcul des tempos, etc.

Nous commençons par présenter le contexte particulier du développement logiciel d'*Antescofo*, suivi par l'organisation générale du code. Nous détaillons ensuite les problèmes posés par l'analyse syntaxique, l'implémentation de la notion de coroutine, qui structure l'exécution, puis la dynamique du moteur d'exécution. Nous présentons enfin les techniques utilisées pour l'évaluation des expressions et la gestion des environnements.

## 8.1 CONTEXTE DU DÉVELOPPEMENT LOGICIEL

*Antescofo* est un système utilisé pour des performances publiques, tant à l'IRCAM qu'en dehors, en France et à l'étranger. Cela impose plusieurs contraintes sur le développement du système :

- La syntaxe du langage doit rester complètement compatible avec toutes les versions antérieures.
- les ressources temps et mémoire utilisées par *Antescofo* doivent rester aussi faibles que possible, car le système prend place dans un environnement complexe ou plusieurs processus critiques se déroulent en parallèle (synthèse sonore en temps réel, spatialisation, etc.).
- Le système doit s'intégrer dans des pratiques de travail (*workflow*) bien établies. Il doit répondre à des besoins spécifiques à chaque étape du travail musical : composition, réalisation de l'électronique, répétition, concert. Cela nous a amené à développer tout une série de commandes qui sont des actions atomiques du langage mais qui relèvent plutôt de l'interface du système et de son utilisabilité.
- Il y a actuellement quatre versions du système qui doivent offrir les mêmes fonctionnalités du point de vue de la machine réactive : objet pour l'environnement Max, objet pour l'environnement PD, programme autonome et librairie (pour l'interface graphique *Ascograph* ou le système de génération de test [CGC14]).

- Toutes ou parties de ces quatre versions doivent être produites pour des environnements systèmes divers : Max 5, Max 6, Max 7, sous MacOS 10.7 à 10.10, plusieurs systèmes Linux/POSIX et le système Windows).
- Il est nécessaire de répondre très vite aux demandes des utilisateurs quand celles-ci portent sur un problème bloquant dans une situation de concert.

Ces contraintes ont amené à l'architecture logicielle présentée dans la section suivante. Elles imposent aussi une structuration solide des développements logiciels, structuration qui doit permettre de gérer simplement les versions alternatives, le prototypage les extensions et les solutions temporaires de contournement des bugs, tout en permettant une gestion distribuée entre les différents intervenants.

En conséquences, les développements se font en `C++`, et sont gérés sous GIT à travers une forge qui permet aussi de gérer la correction des bugs [1]. Le développement se fait principalement dans l'environnement de développement *Xcode* sur Mac, mais avec un processus de configuration réalisé par des scripts *shell* et *Cmake*. Ce processus permet de gérer les dépendances aux nombreuses librairies [2] et aux outils externes [3] qui sont utilisées. Les interactions avec les utilisateurs se font directement pour ce qui est des productions développées à l'*Ircam*, et via la forge et un forum dédié, pour les utilisateurs et les productions en dehors de l'*Ircam*.

## 8.2 ORGANISATION GÉNÉRALE DU CODE

On peut diviser le code en grandes fonctions :

- l'analyseur syntaxique qui convertit un programme Antescofo en structures de donnée et de contrôle *C++* ;
- le modèle de la partition qui associe les actions aux événements ;
- la machine d'écoute qui permet de suivre la position du musicien ;
- le modèle de tempo pour l'estimation des différents tempos en temps réel ; il est utilisé via l'environnement global par la machine d'écoute et l'ordonnanceur ;
- les structures statiques des actions ;
- les coroutines correspondant aux structures d'exécution des actions ;
- l'ordonnanceur ;

1. En deux ans, environ 275 tickets ont été ouverts, dont 91% ont été résolus. Un sondage rapide montre que le temps de résolution est inférieur à la semaine.

2. On peut citer : le calcul de la FFT, *libsndfile* pour l'accès à des fichiers sons, les API de Max ou de PureData, la communication via le protocole OSC, la découverte de service par *Bonjour*, la librairie standard `C++` qui diffère pour chaque version de système et de compilateur, etc.

3. On peut citer *flex* pour l'analyse lexicale, *bison* pour l'analyse syntaxique, les compilateurs `C++` (clang ou g++ suivant la plateforme), etc.

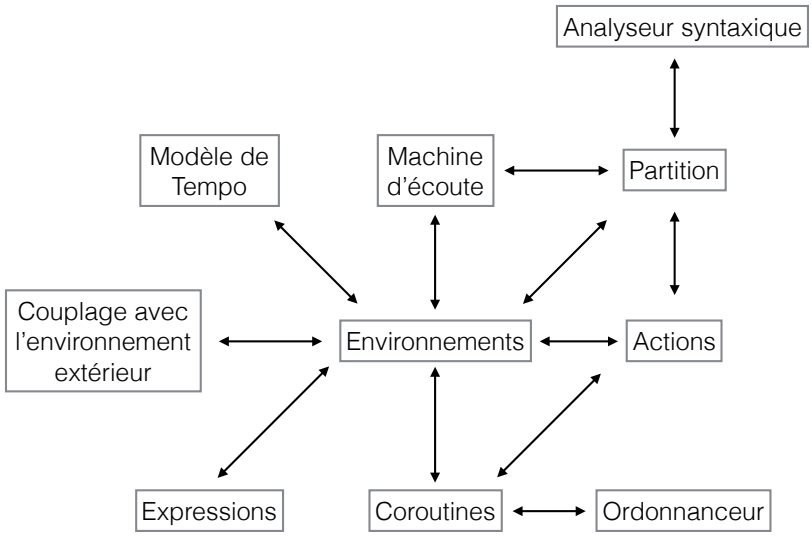



FIGURE 34: Interaction entre les différents champs d'action de l'architecture d'*Antescofo*

- la gestion des expressions ;
- l'environnement global qui fait le lien entre les différentes sous parties auquel il faut ajouter les environnements locaux ;
- le couplage avec l'environnement extérieur audio, réception de commande, envois des message, etc..

Ce code se compartimente en trois sections :

1. la représentation statique d'une partition augmentée ;
2. la représentation des structures dynamiques pour l'évaluation et l'exécution de la partition augmentée, indépendemment de l'environnement hôte ;
3. les fonctions spécifiques à l'environnement hôte : Max, PD ou exécutable autonome.

Certaines entités logicielles traversent les trois sections, comme par exemple le point d'entrée du système. Dans ce cas, ces entités correspondent à des instances de classes qui dérivent les unes des autres ou alors à des méthodes de classes qui ont des réalisations différentes suivant le contexte.

La table 2 donne quelques éléments quantitatifs sur l'organisation de l'architecture logicielle [4].

## 8.3 ANALYSE SYNTAXIQUE

Les analyses lexicales et syntaxiques d'un programme écrit avec le langage Antescofo sont réalisées à l'aide de *lex* et *bison*. L'utilisation

4. Pour donner un ordre de grandeur de la taille du code, l'ensemble des fichiers sources représente 75 000 lignes (comptage par `wc`). L'outil `cloc` reporte 47 000 lignes de code et 10 000 lignes de commentaires dans 113 fichiers.

de ces outils facilitent les modification fréquentes de la syntaxe et l'ajout de nouvelles fonctionnalités dans le langage. Des mécanismes tels que la macro-expansion au vol ou l'inclusion de fichiers multiples, sont développés au-dessus des analyseurs.

*Antescofo* s'insère dans une pratique déjà existante de la musique mixte, c'est pourquoi la syntaxe du langage a été pensée pour faciliter l'intégration d'*Antescofo* dans des environnements tels que *Max* et *PureData*, ce qui permet une prise en main rapide de l'environnement. Par exemple, il n'y a pas de séparateur de messages autre que la fin de ligne. Le langage se développant de manière incrémentale, il est parfois nécessaire de faire quelques compromis pour à la fois satisfaire les contraintes des utilisateurs, assurer la rétro-compatibilité et proposer de nouvelles constructions.

Plusieurs caractéristiques rendent l'analyse lexicale et la grammaire assez complexe à gérer. Les identificateurs de commandes Max et les labels peuvent comporter des caractères non-alphabétiques (ponctuations, symboles d'opérations et de comparaisons, etc.). Les arguments de commandes peuvent être des expressions mais il n'y a pas de séparateurs et la terminaison de la commande est signifiée par la fin de ligne (mais la fin de ligne ne doit pas être significative ailleurs). La macro-expansion se fait au vol, pour éviter plusieurs passes d'analyse. C'est une macro-expansion « à la cpp » et il est par exemple possible de créer de nouveaux identificateurs comme résultat de l'expansion. Une propagation de constante se fait aussi au vol, afin de minimiser les calculs effectivement réalisés à l'exécution et étendre la portée des analyses statiques ultérieures. La prise en charge de ces contraintes amène par exemple à changer l'état de l'analyseur lexicale depuis l'analyseur syntaxique (par exemple pour rendre la fin de ligne significative ou pas). Cependant, nous avons réussi à garder une grammaire LALR(1) non-ambigüe.

Les structures construites lors de l'analyse d'un programme *Antescofo* sont redondantes afin de simplifier autant que faire se peut l'accès aux informations lors de l'exécution. Ces structures sont utilisées lors de l'évaluation d'un programme, mais aussi pour le système de test et pour l'interface graphique *Ascographe*.

Une fois l'arbre syntaxique construit, une passe d'analyse statique permet de lier toutes les références à leur définition (fonctions, processus, labels…), de déterminer le statut de chaque identificateur de variable (variable globale, locale, paramètre de fonction…) et son contexte associé (scope), d'expliciter le contexte temporel de chaque action, etc.

## 8.4 COROUTINES

Une coroutine est l'entité logicielle qui correspond à l'exécution d'une action composée A. C'est un interprète spécialisé qui réalise

l'enchaînement des actions composant A suivant le type de A : par exemple, la coroutine associée à un groupe correspond à une exécution séquentielle d'attente de délais et d'actions.

Contrairement à une fonction, les appels successifs à une routine n'exécutent pas le même code : quand une routine rend la main, elle suspend son exécution logique en un point p bien défini et un appel ultérieur reprendra l'exécution au point p. Dans l'exemple de la coroutine associée à un groupe, les points de suspension correspondent à l'attente d'un délai.

GESTION DE L'ENVIRONNEMENT. C'est dans le contexte d'une coroutine que sont évaluées les expressions. Elles gèrent donc la pile des environnements et installent le contexte correct pour l'évaluation des variables à chaque activation. Ce contexte peut changer au cours de l'activation d'une coroutine, puisque les expressions de tempos ou les conditions de terminaison ne sont généralement pas évaluées dans le même contexte que celui des délais et que les coroutines correspondant à l'exécution des actions filles peut changer à son tour le contexte. La stratégie adoptée est que c'est la coroutine appelée qui est responsable de positionner le bon environnement.

Nous reviendrons sur la gestion des environnements et des variables dans la section suivante.

DES COROUTINES COOPÉRATIVES. L'exécution des coroutines est *a priori* séquentielle : idéalement, l'exécution d'une coroutine est exclusif de l'exécution des autres coroutines et l'ordonnancement des coroutines actives est *coopératif* : il n'y a pas de préemption (une routine a la main jusqu'à ce qu'elle la rende).

Ce modèle séquentiel est effectivement celui qui est réalisé quand l'environnement hôte est PureData ou bien l'exécutable standalone. Mais ce n'est pas le cas pour Max : en effet, à partir de Max6, des threads différents sont utilisés pour le calcul audio et les calculs liés au contrôle. La situation est compliquée car il n'y a pas la même configuration de threads qui est utilisée suivant les options d'exécution choisies sous Max et l'architecture du patch.

Cependant, on peut répertorier les points d'activation des coroutines qui peuvent potentiellement être en concurrence. Ce sont les activations qui interviennent à l'issue de l'attente d'un délai et toutes les interactions avec l'environnement hôte : l'affectation externe d'une variable via la commande `setvar`, les commandes déclenchées par un message, la réception d'un message OSC, etc. La concurrence potentielle entre ces différents points d'entrée est gérée par un verrou qui prend en compte la structure de thread, à la manière de la stratégie `PTHREAD_MUTEX_RECURSIVE` des threads POSIX.

Les coroutines implémentées dans *Antescofo* sont légères et ne dépendent pas du système d'exploitation. Elle sont réalisées par des ob-

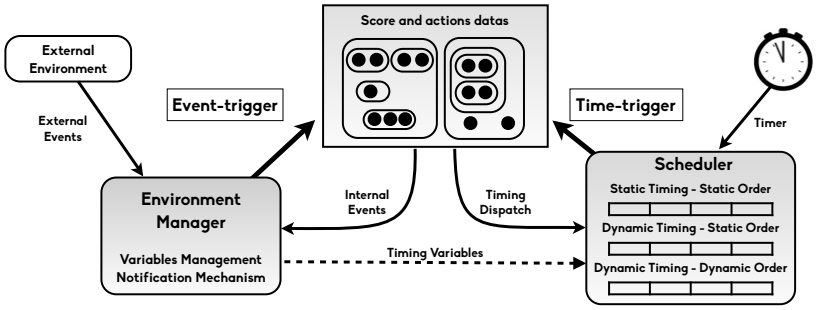



FIGURE 35: Moteur d'exécution d'Antescofo. Des notification correspondant par exemple à la détection d'un événement ou à la mise à jour d'une variable, sont envoyées aux structures statiques et dynamiques associées aux actions via le gestionnaire des environnements. Ces structures font appel à l'ordonnanceur si l'exécution de leur prochaine action doit avoir lieu après l'écoulement d'un délai (absolu, relatif au tempo global ou relatif à un tempo local)

jets dont les méthodes correspondent au code devant s'exécuter entre deux points d'activation. Ces objets sont les éléments manipulés par le moteur d'exécution.

## 8.5 MOTEUR D'EXÉCUTION

Le moteur réactif d'*Antescofo* est construit sur une architecture à la fois événementielle et temporelle, illustrée par la figure 35.

### 8.5.1 *Notification d'événement*

Dans la machine réactive, toute propagation des informations issues de la machine d'écoute ou de l'environnement extérieur se fait via un mécanisme de notification de changement de valeurs d'une variable. Le système maintient à jour, pour chaque variable, une liste d'actions à notifier en cas de changement.

Par exemple, les actions d'un groupe `@tight`, dynamiquement liées à un événement, sont notifiées à travers la variable `$BEATPOS` de la position courante du musicien pour s'exécuter au moment opportun. Ce mécanisme de notification est aussi à la base du fonctionnement de la structure `whenever`. Elle est notifiée à chaque fois qu'une des variables apparaissant dans la condition est mise à jour. Autre exemple, l'ordonnanceur est notifié via le gestionnaire des environnements, des changements de valeurs du tempo et des affectations de variables apparaissant dans les expressions des tempos locaux.

Ce système de notification fait apparaître des problèmes de causalité bien connus dans la communauté synchrone. Il arrive que la notification d'une variable lance des calculs qui résultent, directement ou indirectement, en l'affectation de cette même variable (court-circuit

temporel). Pour stopper le cycle, on inhibe l'exécution d'un bloc si celui-ci a déjà été exécuté dans le même instant logique.

### 8.5.2 *Gestions des différents temps et ordonnancement*

À un instant donné, plusieurs séquences d'actions, s'exécutant en parallèle, peuvent attendre l'expiration d'un délai pour exécuter l'action suivante. *Antescofo* s'appuie sur le service de temps offert par l'environnement hôte (Max-MSP, PureData, POSIX ou IOS) pour compter le passage du temps.

Il n'est pas rare qu'au cours d'une pièce de musique mixte, *Antescofo* coordonne plusieurs centaines d'actions afin de contrôler des processus extérieurs (filtres, spatialisateurs, synthétiseurs, etc.). Il est donc très important de réduire autant que possible les ressources systèmes utilisés par *Antescofo*. Afin d'éviter l'utilisation d'une horloge pour chaque ordonnancement d'une action précédée d'un délai, trois files correspondant à trois types d'évolutions temporelles différentes ont été mis en place dans l'ordonnanceur.

TEMPS STATIQUE - ORDRE STATIQUE : la première file gère des actions dont le délai est exprimé en temps absolu (en secondes). Une fois inséré à la bonne place, l'ordre de réveil des actions et le temps à attendre seront inchangés. Les actions avec des délais exprimés relativement à un tempo dont l'expression est constante sont également gérées dans cette file, la conversion des pulsations en secondes étant faite une fois pour toutes à l'insertion.

TEMPS DYNAMIQUE - ORDRE STATIQUE : la deuxième file correspond aux actions dont les délais sont exprimés en pulsations et qui ne dépendent que du tempo global. La valeur de ces délais définit l'ordre de la file qui ne dépend pas des changements de tempo. Par contre ces changements sont pris en compte pour le recalcul du temps à attendre (en secondes) du premier élément de la file correspondant au délai le plus petit.

TEMPS DYNAMIQUE - ORDRE DYNAMIQUE : la troisième file correspond aux délais qui dépendent soit d'un tempo local exprimé sous la forme d'une expression qui n'est pas une constante (où l'on trouve au moins une variable) soit d'une stratégie d'adaptation automatique du tempo. Les délais convertis en secondes définissent l'ordre de la file. Si une variable est mise à jour alors on réordonne toutes les actions dont le tempo dépend de cette variable.

*Antescofo* n'utilise au final qu'une seule horloge dont la valeur d'initialisation est la valeur minimale des délais en tête des trois files. Si cette valeur est modifiée avant l'expiration de l'horloge alors celle-ci est arrêtée et ré-initialisée à la nouvelle valeur minimale. Si une des

deux autres valeurs est modifiée à une valeur inférieure au temps restant alors l'horloge est ré-initialisée à la nouvelle valeur. Les délais sont enregistrés relativement au précédent, donc seul la mise à jour du délai de tête suffit.

L'approche proposée ici minimise à la fois le coût d'ordonnancement et les dépendances avec l'environnement hôte. Elle permet de contrôler l'ordre entre les différentes actions et structures hiérarchiques partageant un même référentiel temporel.

### 8.5.3 *Évaluation des expressions et gestions des environnements*

Le moteur d'évaluation des expressions a été conçu afin de minimiser les allocations mémoire dynamiques. L'évaluation d'expressions scalaires (entiers, flottants, booléens) ne requiert aucune allocation sur le tas (*malloc*). Les valeurs correspondent à des structures de données mutables (comme les vecteurs) qui sont alloués dynamiquement et désalloués automatiquement via un récupérateur mémoire à compteur de références. Cette approche est nécessaire car il n'est pas possible d'interrompre les calculs pour effectuer une phase de collecte des zones mémoires à récupérer. Le comptage des références permet de libérer la mémoire au plus tôt et fragmente le temps nécessaire à la gestion mémoire en unité tolérable vis à vis des contraintes temps réel.

L'accès aux données se fait via les variables locales ou globales. La portée des variables est lexicale, ce qui permet un accès efficace à la pile des environnements où chaque environnement est associé à une coroutine.

Un environnement est créé quand l'instance d'une action est lancée mais peut survivre à la terminaison de cette action. En effet, les actions filles peuvent référer à une variable locale introduite par leur mère, même après la terminaison de celle-ci. Là aussi, une désallocation automatique via un compteur de référence permet de gérer au mieux la ressource mémoire.

L'héritage du tempo et des stratégies de synchronisation pose des problèmes particuliers car il peut correspondre à une portée dynamique des variables. C'est le cas par exemple pour les processus qui, par défaut, héritent des paramètres liés au tempo de la coroutine dans lequel l'appel est réalisé. Ce tempo n'est pas encore connu au moment de leur définition et donc nécessite un mécanisme dynamique pour récupérer la valeur d'une variable utilisée dans ces paramètres. Un attribut (`@static`) permet cependant un héritage statique de la temporalité d'un processus.

L'accès dynamique à la valeur d'une variable est également réalisé à travers la notation pointée. Ainsi, l'expression `$coroutine.$x` permet d'accéder à la valeur de `$x` dans l'environnement de la coroutine pointée par la valeur de la variable `$coroutine` et ceci quelque soit

l'endroit où se situe cette expression. Ce genre de mécanisme s'avère très utile pour permettre à l'utilisateur de communiquer et interagir avec les coroutines, un peu à la manière d'une programmation objet.

La gestion des variables dans les fonctions reste simple car les fonctions sont définies globalement et ne créent pas de portée ce qui implique que seul l'accès aux variables globale est possible. Les fermetures ne concernent que les applications partielles où il suffit de sauvegarder la valeur des arguments déjà fournis. Ce fonctionnement est simple à gérer et ne nécessite aucun mécanisme de ramasse-miettes. facilitant la mis en place d'une bibliothèque riche avec des fonctions d'ordre supérieur ( `map`, `fold`, etc.).

| Fonction | Classes | Fichiers (.x = .h et .cpp) |
|---|---|---|
| Analyse Syntaxique | *parsing* | *4 500 loc* : parser.yy, scanner.lll, parsedriver.x |
| | *Représentation des actions* `class Action` qui se raffine en 75 sous-classes | *8200 loc* : Action.x, ActionCompound.x, ActionAtomic.x, ActionDsp.x ActionFire.cpp |
| | *Représentation des événements* `class Event` qui se raffine en 6 sous-classes | *1 000 loc* : Event.x |
| | *Représentation des expressions* `class Expression` qui se raffine en 35 sous-classes | *13 000 loc* : Expression.x, Function.x, IFunction.x, Function_*uuu*.cpp |
| | *Représentation des valeurs* `class TaggedValue` qui se raffine en 13 sous-classes | *3 000 loc* : Value.x Interpolation.cpp |
| | *Représentation de la partition* `class Score` | *2 500 loc* : Score.x |
| Machine d'écoute et modèle de tempo | `class AnteAlign, StateChain, t_tempo` | *4 500 loc* : AnteAlign.x StateChain.x TempoClass.x |
| coroutine | `class exe` et 7 sous-classes | *3 200 loc* : exec.x, Proc.x |
| ordonnanceur et services d'horloge | `class t_scheduler,` | *4 000 loc* : AnteScheduler.x, ActionFire.cpp, AnteClock.x, AnteClock_externals.cpp, AnteClock_stabdalone.cpp |
| gestion de l'environnement | `class Environment` et `GlobalEnvironment` | *3 200 loc* Environnement.x, Environment_RunTime.cpp |
| liaison à l'environnement hôte | Max | *4 300 loc* Antescofo_Max.cpp, Antescofo_MaxSDK.x |
| | PureDate | *4 300 loc* Antescofo_Puredata.x |
| | standalone | *2 300 loc* Antescofo_standalone.x |

TABLE 2: Organisation logicielle de l'implémentation actuelle d'*Antescofo* en trois compartiments (représentation statique des entités d'un programme, *run-time*, liaison au service de l'hôte) et six grandes fonctions.
Les chiffres données sont approximatifs mais donnent une idée correcte des ordres de grandeur des développements logiciels.
Un certains nombre de fonctions ne sont pas répertoriées ici : la liaison à l'environnement d'édition et de suivi graphique *Ascograph*, la gestion de l'OSC, la compilation des motifs temporels pour les `whenever`, la gestion des calculs DSP, etc.

Troisième partie

# ANTESCOFO DANS LA PRATIQUE

Cette troisième partie correspond aux études que nous avons menées pour valider les mécanismes proposés dans le chapitre précédent et les implémentations correspondantes. Les constructions présentes sont-elles utiles et musicalement pertinentes ? Nous croyons qu'il ne peut pas y avoir de réponses complètement formelles à cette question. On peut vérifier si un programme possède certaines propriétés, si une implémentation répond à des contraintes de ressources mais pour savoir si un langage se révèle adapté à l'expression de la pensée de ses utilisateurs, il faut le confronter à des applications réelles. C'est pourquoi les études réalisées dans cette partie représentent un grand volume de travail.

Nous présentons ici huit études :

- Le chapitre 9 propose plusieurs implémentations de l'œuvre *Piano Phase* de Steve Reich. L'objectif est de donner une vue d'ensemble du langage et d'illustrer ses possibilités et sa puissance sur un exemple concret.
- Le chapitre 10 détaille les séances de tests réalisées avec le compositeur Marco Stroppa et le pianiste Florent Boffard pour l'étude et l'évaluation des mécanismes de synchronisation du langage ;
- Les développements pour les applications d'accompagnement automatique seront présentés au chapitre 11. Ils ont été testés en collaboration avec *l'Orchestre de Paris*.
- Le chapitre 12 décrit le travail réalisé au cours d'une résidence en recherche artistique, sur la réalisation en temps réel de canons rythmiques. Au-delà de cet exemple musicalement important, cette application a initié l'étude des stratégies d'anticipation qui ont donné lieu aux mécanismes de synchronisation, utilisés aujourd'hui dans des applications plus simples d'accompagnement automatique.
- Les chapitres 13 et 14 montrent l'utilisation du langage dans la composition de deux œuvres de musique mixte qui requièrent un contrôle *fin* pour la synthèse de nombreux processus sonores et qui reposent sur des mécanismes *dynamiques*.

- Le développement d'un logiciel dans le cadre des musiques improvisées est présenté dans le chapitre 16. Cet exemple montre qu'*Antescofo* peut être utilisé dans un contexte de partition ouverte ou de scénario d'improvisation. Le système est ici utilisé comme un oeil permettant d'implémenter la coordination des différents modules du système.
- Enfin le chapitre 17 introduit l'utilisation d'*Antescofo* pour la création du dispositif interactif utilisé par un groupe de rock électronique dans leur concert.

# 9

# QUATRE ÉCRITURES DE *PIANO PHASES*

Nous proposons d'illustrer les fonctionnalités du langage à travers quatre programmes correspondant tous à la même pièce : *Piano Phase* du compositeur Steve Reich, initialement prévue pour deux pianos. Cette pièce est construite sur la répétition d'une séquence de douze notes, jouée par les musiciens $M_1$ et $M_2$ en parallèle, où $M_2$ se déphase par rapport à $M_1$. Le processus compositionnel peut être décrit algorithmiquement de la façon suivante, en adéquation avec la partition originale (figure 36) : Pendant les $n_p$ premières périodes ($12 \leqslant n_p \leqslant 18$), $M_1$ et $M_2$ jouent les mêmes notes au même moment. L'étape de déphasage peut alors commencer : $M_2$ accélère légèrement, se déphase par rapport à $M_1$ jusqu'à ce que le déphasage atteigne après $n_d$ périodes ($4 \leqslant n_d \leqslant 16$), une valeur équivalente à une double-croche. À ce moment, les deux voix se resynchronisent et $M_2$ reprend le tempo initial (celui de $M_1$). La superposition des notes formant une nouvelle couleur musicale pendant $n_p$ périodes. Ces deux étapes sont ensuite répétées douze fois, exécutant les douze superpositions possibles. La pièce se finit sur la première étape à l'unisson. Dans tous nos exemples, nous prendrons $n_p = n_d = 4$.

Le premier programme est autonome et extensionnel. Le deuxième programme est équivalent au précédent mais il est cette fois-ci intentionnel. Les troisième et quatrième versions sont interactives. La troisième utilise un enregistrement pour séquencer la voix qui se déphase tandis que la quatrième réinjecte la propre performance du musicien pour créer le déphasage.

## 9.1 CONTRÔLE DYNAMIQUE DES DÉLAIS

Dans la première version présentée ci-dessous, deux boucles électroniques remplacent les deux musiciens. Chacune d'elles correspond à une voix qui séquence des messages envoyés à un synthétiseur. Les actions jouées par le synthétiseur sont définies dans le corps de boucle avec une période de trois pulsations pour la première et une

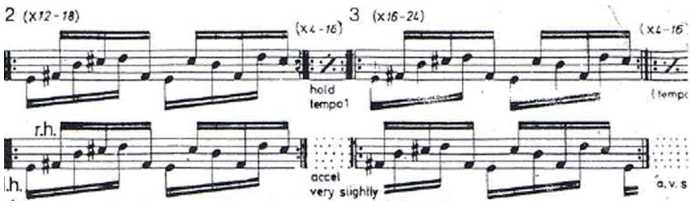



FIGURE 36: Extrait d'une partition de *Piano Phase* du compositeur Steve Reich

période définie par `$period` pour LoopDec, la seconde. Cette dernière variable est initialisée à trois pulsations, la durée d'une mesure. Les délais entre chaque action de la première boucle valent $1/4$ de pulsation ; pour la deuxième boucle, ils sont spécifiés par la variable `$delay` (initialisée à une double-croche). La variable `$cpt` permet de repérer le moment où l'on passe d'une étape de déphasage à une étape en phase. Le compteur est incrémenté dans la boucle. Une fois que la valeur 4 est atteinte, on diminue la période et les délais. Lorsque le compteur est égal à 0 modulo $(4+4)$, la période et les délais sont remis à leurs valeurs initiales.

La fonction `beat2ms` est une fonction auxiliaire qui permet de convertir des pulsations en secondes pour transmettre la durée d'une note au synthétiseur. `mnote1` et `mnote2` sont le noms des receveurs correspondant au deux synthétiseurs.

```
@fun_def beat2ms($X) {
        1000*$X*60/$RT_TEMPO
}
   $periode := 3
   $delai := 1/4
   $compteur := 0
   antescofo::tempo 100

   loop LoopRef 3 {
      mnote1 64 120 @beat2ms(1/4)
      1/4 mnote1 66 120 @beat2ms(1/4)
      1/4 mnote1 71 120 @beat2ms(1/4)
      1/4 mnote1 73 120 @beat2ms(1/4)
      1/4 mnote1 74 120 @beat2ms(1/4)
      1/4 mnote1 66 120 @beat2ms(1/4)
      1/4 mnote1 64 120 @beat2ms(1/4)
      1/4 mnote1 73 120 @beat2ms(1/4)
      1/4 mnote1 71 120 @beat2ms(1/4)
      1/4 mnote1 66 120 @beat2ms(1/4)
      1/4 mnote1 74 120 @beat2ms(1/4)
      1/4 mnote1 73 120 @beat2ms(1/4)
}

loop LoopDec $periode {
   mnote2 64 120 @beat2ms(1/4)
   $delai mnote2 66 120 @beat2ms($delai)
   $delai mnote2 71 120 @beat2ms($delai)
   $delai mnote2 73 120 @beat2ms($delai)
   $delai mnote2 74 120 @beat2ms($delai)
   $delai mnote2 66 120 @beat2ms($delai)
   $delai mnote2 64 120 @beat2ms($delai)
   $delai mnote2 73 120 @beat2ms($delai)
   $delai mnote2 71 120 @beat2ms($delai)
   $delai mnote2 66 120 @beat2ms($delai)
   $delai mnote2 74 120 @beat2ms($delai)
   $delai mnote2 73 120 @beat2ms($delai)
```

```
   $compteur := ($compteur+1) % 8

   if ($compteur = 4) {
      $periode := $periode-(1/4)/4
      $delai := $periode / 12
   }
   if ($compteur = 0) {
      $periode := $periode+ (1/4)/4
      $delai := $periode / 12
   }
}
```

## 9.2 CONTRÔLE DE DYNAMIQUE DU TEMPO + PROCESSUS

Le deuxième exemple est équivalent au premier. Cette fois-ci on a défini un processus récursif qui va jouer une note puis se rappeler lui-même après un certain délai pour jouer la note suivante. Les notes à jouer sont stockées dans les tableau `$notes`.

Le déphasage est contrôlé à travers les variations de tempo du groupe englobant l'appel du processus (liaison dynamique des paramètres temporels).

Le changement de tempo est réalisé à l'aide d'un `whenever` qui « écoute » la variable globale `$cmpt`, modifiée par les processus correspondant à la deuxième voix.

La valeur de la variable `$i` (indice courant du tableau), locale à chaque voix, est accessible en liaison dynamique grâce à la commande `$MYSELF` permettant de récupérer un pointeur sur la coroutine courante.

La fonction `@command` permet d'évaluer une expression pour définir le nom du receveur.

```
@proc_def ::rec_proc($ind) {
   @command("mnote"+$ind) ($notes[$MYSELF.$i]) 120 (1000*1/4*60/$
       local_tempo)
   $MYSELF.$i := ($MYSELF.$i + 1)%12
   if($ind = 2){
        $cmpt := ($cmpt + 1) % 96
   }
   1/4 ::rec_proc($ind)
}

$notes := [64,66,71,73,74,66,64,73,71,66,74,73]
$cmpt:=-1
$tempo1:=100
$tempo2:=$tempo1

group @tempo = $tempo1 {
        @local $i
        $i := 0
```

```
        ::rec_proc(1)
}

group @tempo = $tempo2 {
        @local $i
        $i := 0
        $proc2 := ::rec_proc(2)
}

whenever($cmpt = 48) { //dephasage
        $tempo2 := $tempo1 * (3*4+1/4)/(3*4)
}
whenever($cmpt = 0) {  //en phase
        $tempo2 := $tempo1
}
```

## 9.3 AVEC SUIVI DE PARTITION ET CONTRÔLE CONTINU D'UN FLUX AUDIO

Dans le troisième exemple on synchronise un accompagnement avec le jeu d'un musicien. Le musicien joue la voix de référence tandis que l'électronique joue la voix qui se décale. Le jeu du musicien est suivi par la machine d'écoute qui détecte les notes jouées. On utilise le vocodeur de phase *SuperVPScrub~* pour contrôler la position d'un fichier audio dans lequel la boucle de 3 pulsations a été préalablement enregistrée. On déphase le fichier audio par rapport au musicien en contrôlant la période comme précédemment.

```
   $periode := 3
   $compteur := 0

NOTE 64 0.25 begin
   loop l $periode @target [6s]
   {
      ScrubPos 0.
      abort c
      curve c
         @Grain := 0.05 s,
         @Action := ScrubPos ($x * 1000) (0.05 * 1000)
      {
         $x
         { { 0.} $periode { 1.4912 } }
      }
      $compteur := ($compteur+1) % 8

      if($compteur = 4)
      {
         $periode := $periode - (1/4)/4
      }

      if($compteur = 0)
```

```
      {
         $periode := $periode + (1/4)/4
      }
   }
NOTE 66 0.25
NOTE 71 0.25
NOTE 73 0.25
NOTE 74 0.25
NOTE 66 0.25
NOTE 64 0.25
NOTE 73 0.25
NOTE 71 0.25
NOTE 66 0.25
NOTE 74 0.25
NOTE 73 0.25 @jump begin
```

## 9.4 AVEC SUIVI D'UNE PULSATION ET RÉINJECTION

Le dernier exemple est également une situation où le musicien joue la partie de référence et *Antescofo* contrôle un processus correspondant à la voix qui se déphase. La synchronisation n'est cependant pas réalisée grâce à un suivi des événements du musicien mais à partir d'une information de pulsation envoyée depuis l'environnement extérieur (pédale, détecteur de beat). Cette information est récupérée via les mises à jour de la variable `$var_sync`. De plus, l'accompagnement correspond ici à une réinjection de la performance du musicien enregistré en temps réel. Comme précédemment, une curve envoie toutes les 0.05 seconde la position du fichier qu'il faut lire au vocodeur de phase. Cet accompagnement démarre avec 3 pulsations de retard par rapport au temps-réel et rattrape le temps-réel dans les périodes de déphasage. Le déphasage est réalisé en diminuant la valeur de `$puls`, accélérant ainsi la lecture d'une pulsation.

```
$compteur:=0
@tempovar $var_sync(100,1)
$tab := []
$ind_ref := -1
$ind_dec := - 1

WHENEVER($var_sync)
{
   $compteur := $compteur + 1
   if($compteur = 1)
   {
        $start := $NOW
                start_recording 1
   }
   $ind_ref := ($ind_ref +1) % 6

   $tab[$ind_ref] := $NOW - $start
```

```
   if($compteur = 3)
   {
        loop l $puls @target [4s] @sync $var_sync
        {

           $ind_dec := ($ind_ref +1) % 6
           if(($compteur%24) = 12 )
           {
                $puls := ((1/4)/4)/3
           }
           if(($compteur%24) = 0 )
           {
                $puls := 1
           }
           abort c
           curve c @Grain := 0.05 s , @Action := ScrubPos ($x *
               1000) (0.05 * 1000)
           {
                $x
                {
                                { ($tab[$ind_dec])}
                        $puls { ($tab[($ind_dec+1)%6]) }
                }
           }
        }
  }
}
```

# 10

# ÉTUDE PRATIQUE DES STRATÉGIES DE SYNCHRONISATION

Plusieurs séances de travail ont été organisées à l'*Ircam* avec le pianiste Florent Boffard et le compositeur Marco Stroppa afin d'étudier en pratique la coordination dans *Antescofo* des actions électroniques avec le jeu d'un musicien. L'objectif de ces séances était multiple :

- évaluer les mécanismes du langage pour la gestion du temps (stratégies de synchronisation, calculs du tempo, etc.) ;
- trouver les structures de contrôle du langage les plus adaptées pour chacun des objets contrôlés, afin de réaliser un accompagnement à la fois musical et avec un rendu sonore d'une qualité optimale ;
- recueillir les impressions d'un pianiste de renommée mondiale sur la synchronisation de processus électroniques par *Antescofo* ;
- permettre au compositeur de mieux comprendre les possibilités offertes par le langage afin de les utiliser au mieux dans ses compositions ;
- déterminer les limites du système et évaluer de nouvelles pistes de recherche.

Nous détaillons dans ce chapitre la mise en œuvre des partitions de tests qui nous ont permis de réaliser ces séances, les différents problèmes rencontrés et les solutions proposées pour contrôler un vocodeur de phase, pallier la latence et adapter les stratégies de synchronisation au contexte musical. En conclusion nous proposons des améliorations du langage pour faciliter encore plus avant la spécification d'une interaction fine et musicale entre le musicien et la machine.

## 10.1 SETUP

L'ensemble des tests ont été réalisés en utilisant la machine d'écoute d'*Antescofo* pour suivre le jeu du pianiste. L'objet *Antescofo* est utilisé dans l'environnement de programmation *Max* qui permet de récupérer le flux audio grâce à un microphone placé sur le piano. *Antescofo* contrôle d'autres objets *Max* qui génèrent du son. Il s'agit d'un lecteur de fichier audio simple et de différentes versions du vocodeur de phase *SuperVP* permettant contrôler la lecture d'un fichier audio en vitesse ou en position sans déformation fréquentielle [RR05].

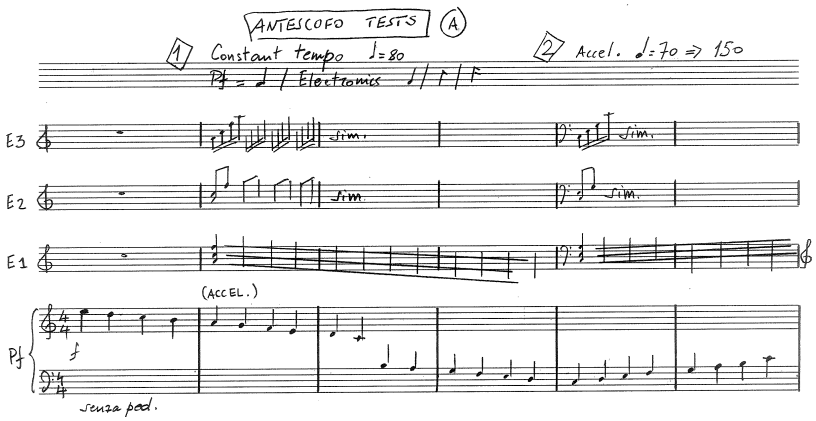



FIGURE 37: Extrait d'une des partitions de tests écrites par Marco Stroppa

### 10.1.1 *Les partitions*

En amont de ces séances nous avons mis en place un protocole répertoriant toutes les combinaisons possibles des différentes valeurs des paramètres temporels. Ces combinaisons ont ensuite été appliquées sur différents types d'interactions, du plus simple accompagnement, à la synchronisation d'oeuvres du répertoire classique.

Les premières partitions correspondent à des gammes jouées en noires, en croches ou en double-croches avec un accompagnement électronique correspondant à des noires, des croches ou des double-croches (cf. Figure 39).

Deux inventions de Johann Sebastian Bach ont été choisies pour mettre en œuvre les différents mécanismes temporels étudiés dans un contexte plus musical. Enfin quelques tests ont également été réalisés sur le *Nocturne* n° 1 de Frédéric Chopin, pièce romantique où les variations expressives du tempo sont extrêmes.

À chaque fois, la partie correspondant à la *main gauche* est coordonnée avec le jeu du musicien qui joue la partie correspondant à la *main droite*.

### 10.1.2 *Les objets contrôlés*

Toutes les parties d'accompagnement ont été préparées en amont de la séance sous la forme de fichiers audio. Différentes stratégies pour la restitution ont été employées :

- Lecture à vitesse originale de courts fichiers sons ; la partition augmentée envoie des messages pour lancer les lectures aux bons moments.
- Lecture des fichiers courts à l'aide de l'objet *SuperVpPlay~* permettant notamment de contrôler la vitesse de lecture de fichiers sons. Le programme *Antescofo* envoie les commandes de position, de vitesse et de démarrage.

– Lecture d'un seul long fichier en envoyant depuis *Antescofo*, une nouvelle position de lecture à l'objet *SuperVpScrub~* toutes les 5 millisecondes.

La spécification de la synchronisation avec *SuperVpScub~* (en position) est plus intuitive que celle en vitesse. Nous détaillons ici la spécification dans le cas d'une synchronisation avec un seul fichier audio pour toute la durée de la pièce.

– Les marqueurs correspondants aux pulsations du fichier audio sont stockés dans un tableau.
– À chaque itération, la construction `curve` C est la construction qui permet d'envoyer la valeur de la position que l'on veut lire pendant la performance. Ces messages sont envoyés toutes les 5 millisecondes, la `curve` interpolant automatiquement entre deux valeurs du tableau pendant une pulsation.
– Une boucle L avec une période d'une pulsation nous permet d'itérer sur les valeurs du tableau.

Voici une version simplifiée du programme :

```
   $marqueurs :=  [0.0,0.75,1.5,... ]
   $i :=0
NOTE 60 1.0
   loop L 1.0 @target [5 s]
   {
      curve C @Grain := 0.05s
         @action := ScrubPos $pos
      {
         $pos
         {
               { ($marqueurs[$i])}
            1 { ($marqueurs[$i+1]) }
         }
      }
      $i := $i+1
   }until(i>@size($marqueurs))

NOTE 62 1.0
         ...
```

Cependant, avec cette dernière solution, nous avons constaté quelques problèmes dans le rendu sonore : les attaques des notes du fichier audio étaient parfois dénaturées. Cela est lié à l'envoi de messages trop rapprochés dans le temps, lorsqu'une nouvelle instance de la boucle est lancée. Différentes solutions sont envisageables pour pallier à ces imprécisions : programmer l'envoi des messages avec une seule `curve` au lieu de plusieurs `curve`s dans une boucle, utiliser deux objets *SuperVpScub~* qui s'alternent avec un effet de fondu enchainé (technique utilisée dans l'application *Improtek* détaillée en chapitre 16). Une autre solution envisageable serait d'introduire dans le langage une notion de grain d'échantillonnage commun à une même hiérarchie de blocs.

Ici on spécifierait ce grain au niveau de la `loop` pour que l'envoi tous les messages se calent sur une grille temporelle.

## 10.2 LE PROBLÈME DE LA LATENCE

La chaîne de traitement (carte son, application hôte, machine d'écoute d'*Antescofo*, objets contrôlés, etc.) engendre une latence qu'il faut compenser pour une meilleure synchronisation de l'accompagnement. Cette latence varie en fonction des objets contrôlés, des paramètres d'analyse, des paramètres d'entrée/sortie, etc.

Deux solutions ont été adoptées pour compenser cette latence. Si elles ont globalement résolu le problème, elles restent à être affinées. La première solution consiste à anticiper le lancement des actions dans la partition *Antescofo* de la façon suivante :

```
//Sans prendre en
//compte la latence


NOTE 60 1.0
NOTE 62 1.0
   group
   {
    a1
    1 a2
     ...
   }
```

⟶

```
//En prenant en
//compte la latence
NOTE 60 1.0
   $lat_abs := 0.1
   $lat_rel := $lat_abs*$RT_TEMPO/60
    (1-$lat_rel) group
    {
    a1
    1 a2
     ...
   }
NOTE 62 1.0
```

Dans cet exemple, le groupe doit être exécuté au moment de la détection du deuxième événement. Pour prendre en compte la latence, le groupe est associé à l'événement précédent pour qu'il soit exécuté après un délai égal à la durée de cet événement moins la valeur de la latence convertie en pulsation. Toute la séquence sera donc anticipée de cette valeur calculée. Ceci n'est cependant pas suffisant si les variations de tempo sont trop grandes. La position courante étant toujours exprimée de manière relative, il n'existe pas de moyen simple dans le langage pour décaler uniformément toute une séquence d'action d'une valeur en temps absolu.

Une deuxième solution facile à mettre en œuvre consiste à ajouter la valeur de la latence à la valeur de la position de lecture envoyée à l'objet, position exprimée en secondes. Cette solution n'est applicable que si les actions envoyées correspondent à des positions exprimée en secondes et elle n'est, qui plus est, pas tout à fait exacte. En effet, il faut aussi prendre en compte à la fois les variations de tempo du fichier audio et le tempo courant du musicien dans le calcul de la latence. Enfin cette solution demande à gérer explicitement le lancement du groupe.

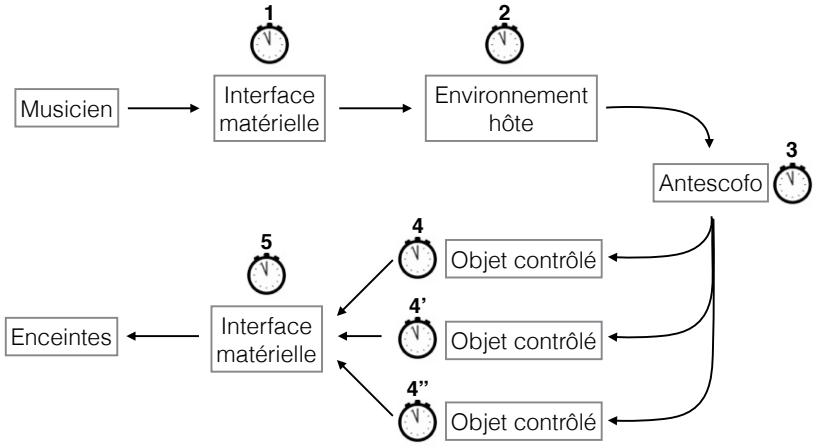



FIGURE 38: Schéma représentant le parcours des informations du musicien jusqu'aux enceintes. Pour être synchrone avec le musicien, les messages envoyés aux objets depuis *Antescofo* doivent être anticipés en fonction de la latence de l'objet contrôlé et de la somme des autres latences de toute la chaîne de calcul.

Même si en pratique ces solutions nous ont permis de résoudre ponctuellement le problème de la latence, il nous paraît essentiel d'introduire des mécanismes génériques permettant la prise en compte d'une valeur de décalage exprimée en seconde pour anticiper l'exécution des actions tout en s'adaptant aux variations du tempo local de chaque bloc (cf. Figure 38). Ces mécanismes sont en cours d'étude.

Lors de ces séances de tests, la valeur de la latence était estimée « à l'oreille », en modifiant dynamiquement les valeurs d'anticipation. Il serait également intéressant de mettre en place des outils de calibration dans le langage facilitant l'estimation de cette valeur pour chacun des objets contrôlés dans la partition.

## 10.3 ESTIMATION DU TEMPO DU MUSICIEN

En utilisant les constructions de calcul de tempo à partir des mises à jour de variables (cf. section 6.5), différentes estimations du tempo ont pu être comparées à l'estimation du tempo de la machine d'écoute d'*Antescofo* (algorithme adapté de Edward Large [LJ99]). Ces estimations sont utilisées pour anticiper le comportement du musicien dans la synchronisation des processus électroniques.

La première variante consistait à calculer du tempo *instantané* (rapport des durées relatives et absolues entre deux événements) sur chaque événement du musicien. Bien que les stratégies anticipatives ont tendance à atténuer les variations de tempos, cela n'était clairement pas suffisant dans le cas de notes rapides.

Les autres variantes consistaient à choisir les événements sur lesquels on calcule le tempo. Cette approche s'est avérée tout à fait

concluante puisque le résultat attendu correspondait à l'idée musicale du compositeur. Cela nous a amené à considérer la possibilité d'introduire dans le langage des annotations simplifiant la spécification de cette information.

## 10.4 LES STRATÉGIES DE SYNCHRONISATION

Les stratégies de synchronisation anticipatives avec cibles dynamiques (cf. section 6.4.2) sont celles qui ont été le plus utilisées pour gérer la temporalité des processus mis en jeu. Deux raisons peuvent l'expliquer :

- elles offrent les propriétés requises pour le contrôle d'un flux continu ;
- elles sont souples à manipuler car la fenêtre de rattrapage peut être modifiée dynamiquement, selon le passage d'une stratégie très réactive (fenêtre de 0) à une stratégie très lâche (grande fenêtre).

Nous envisageons cependant des tests supplémentaires pour mieux maîtriser les stratégies de synchronisation avec pivots statiques. En effet, nous avons remarqué que les variations de re-synchronisation pour les stratégies avec pivots sont parfois trop grandes selon la distance entre la position courante et la prochaine cible. En les combinant avec des stratégies dynamiques, par exemple en prenant à chaque instant la stratégie qui se re-synchronise le plus rapidement avec le musicien, cela pourrait résoudre le problème.

### 10.4.1 *Tempo constant, accelerando, ritardando*

Sur les partitions correspondant aux gammes, nous avons demandé au pianiste de jouer à tempo constant, en accélérant, ou en ralentissant. À tempo constant, une fois les valeurs de latence bien réglées, nous avons pu constater l'influence de la taille de la fenêtre des stratégies dynamiques sur le résultat musical. Le contrôle de cette fenêtre est intéressant et c'est un paramètre sur lequel le compositeur peut facilement jouer. En demandant au pianiste d'intégrer du *rubato* dans son jeu, le pianiste avait le sentiment d'être musicalement bien suivi par la machine même s'il pointait le *manque d'initiative* de la partie électronique.

Cette impression était accentuée lorsque la gamme était jouée avec une évolution constante du tempo. L'accompagnement était toujours *derrière* dans le cas de l'accélération, *devant* dans le cas du ralentissement.

Deux solutions ont été développées pour résoudre ce problème. La première consistait à attribuer directement un tempo pour l'accompagnement. Le calcul du tempo correspondait à une moyenne entre le tempo estimé en temps réel et d'une valeur dont l'évolution était fixée

à l'avance (par exemple de 40 à 180 en 40 pulsations). Cette solution est trop rigide car ne elle s'adapte pas suffisamment aux différentes interprétation du musicien.

La deuxième solution, beaucoup plus satisfaisante, consiste à altérer l'estimation que l'on a du tempo du musicien plutôt que d'agir directement sur le tempo du processus. Cela permet ainsi de tirer parti des stratégies de synchronisation qui prennent en compte la position détectée du musicien. Encore une fois, nous utilisons les constructions permettant d'associer un référentiel calculé (via les mises à jour d'une variable) à l'exécution d'une séquence. Le calcul du tempo estimé dépend à la fois de la valeur de tempo estimée par l'algorithme de Large et d'une valeur dont l'évolution est fixée à l'avance comme précédemment. Nous avons constaté que l'évolution de cette valeur devait être non-linéaire pour mieux suivre l'accélération du tempo.

Voici le le programme correspondant à un accelerendo. La première partie du code correspond au calcul du tempo, la deuxième au séquencement des actions.

```
curve @grain := 0.1s ,
  @action := {
      let $tempo_hyb.tempo := @max($RT_TEMPO, ($RT_TEMPO + $x)
          /2)
    }

  }
  {
    $x{
           {$RT_TEMPO} @type "quad"
      40   {($RT_TEMPO + 100)}
    }
  }
}

group  @target [6 s] @sync $tempo_hyb
{
  actions...
}
```

Ce mécanisme permet à l'accompagnement de mieux suivre l'accélération du musicien. La variable `$x` correspond à un terme de « forçage » qui pallie le manque d'anticipation de l'algorithme original de Large. On obtient un résultat musical qui « fait preuve d'initiative » tout en s'adaptant aux variations de l'interprétation (cf. Figure 39).

Cette technique a ensuite été utilisée de façon concluante pour les passages expressifs des œuvres de Bach et de Chopin (accelerando, ritardando).

Nous sommes actuellement en train d'étudier comment introduire dans le langage des annotations pouvant être utilisées pour spécifier ce genre de variations temporelles.

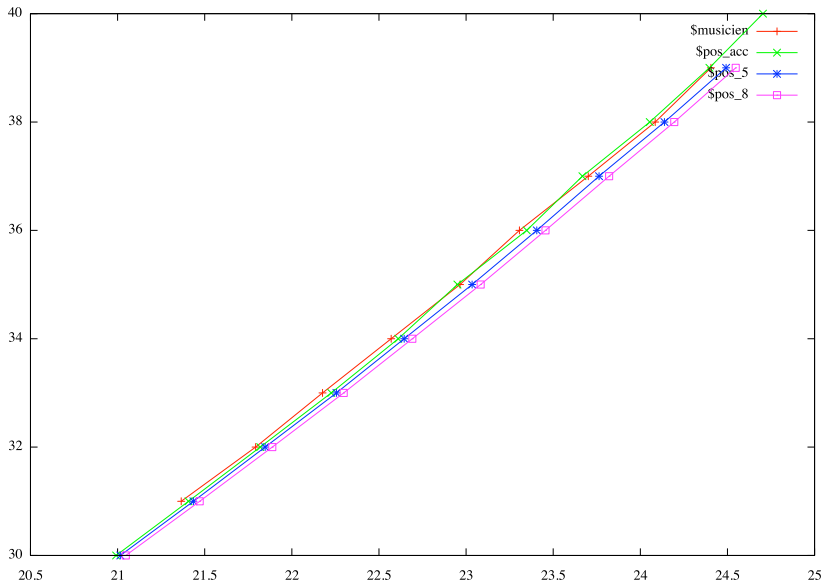



FIGURE 39: Ce graphique montre la temporalité de trois séquences musicales coordonnées par rapport au tempo et aux événements du musicien (croix rouge). Le musicien joue une gamme en accélérant, l'accompagnement entendu correspond à la courbe verte. Le processus associé à cette courbe bénéficie d'une indication supplémentaire sur la variation de tempo par rapport aux deux autres processus (fenêtre de 5 et fenêtre de 8) qui sont sont donnés ici à titre de comparaison. La courbe verte est parfois au dessus de la rouge ce qui illustre la « prise d'initiative » pour l'accélération tout en restant proche de la temporalité du musicien. Les deux autres sont tout le temps en retard.

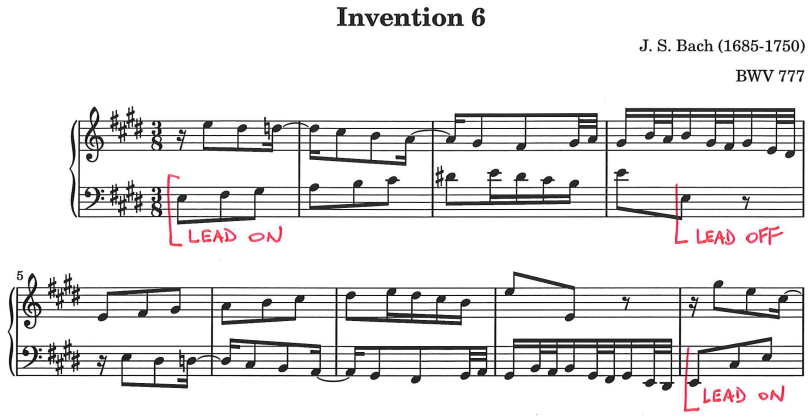



FIGURE 40: Extrait de l'Invention n° 6 de Johann Sebastian Bach. Des fichiers audio correspondant à la main gauche sont synchronisés avec le musicien. Les annotations indiquent quand la machine doit diriger ou au contraire suivre le musicien.

### 10.4.2 *leader/suiveur*

Les études perceptives évoqués dans le chapitre 2, ont montré les relations complexes qui existent entre les musiciens d'une même formation musicale dans la temporalité de leur jeu, en particulier dans la relation leader/suiveur.

Les stratégies de synchronisation font en sorte que les processus électroniques s'adaptent de manière plus ou moins réactive aux variations de l'interprète. Seulement, nous avons rapidement constaté que dans une pièce musicale, chaque voix qui s'exécute en concurrence avec les autres peut prendre à tour de rôle le « lead ».

Toujours en utilisant les constructions permettant de coordonner un processus électronique sur les mises à jour d'une variable, nous avons pu programmer l'alternance entre coordination avec le musicien et tempo fixe (le dernier détecté) pour une même séquence. Lorsque le musicien est maître du tempo, la variable de synchronisation est mise à jour au rythme des événements du musicien. Dans le cas contraire, la variable est mise à jour dans la temporalité de la séquence, ce qui implique un tempo constant dans la séquence des actions.

## 10.5 ALTERNER ESTIMATION PROGRESSIVE ET CONSERVATIVE

Nous avons décrit dans le chapitre 6 les différences entre les stratégies progressives et anticipatives qui correspondent aux estimations du musiciens `$T_NOW` et `$A_NOW`. La stratégie progressive (stratégie par défaut) est satisfaisante en particulier dans les passages avec beaucoup d'événements à reconnaître. Mais la stratégie conservative est

plus précise dans les passage de ritardando. Alterner entre ces deux stratégies en fonction des passages contribue à améliorer le rendu musical de l'accompagnement.

## 10.6 NOCTURNE

En testant le système sur un Nocturne de Frederic Chopin, le but était de réussir à combiner toutes les solutions techniques décrites précédemment dans un contexte extrêmement expressif où les relations temporelles entre la main gauche et la main droite sont complexes. Encore une fois, *Antescofo* contrôle des fichiers audio correspondant à la main gauche pendant que le musicien ne joue que la main droite. Les annotations de Marco Stroppa ont servi à préparer la partition *Antescofo* en utilisant les différentes solutions (cf. Figure 41).

Même si le résultat n'est pas aussi convaincant qu'une performance réelle par un orchestre dirigé par un chef humain, ils restent néanmoins très positifs : un ajustement fin de chaque paramètre permet comportement expressif qui s'adapte aux variations de l'interprétation.

## 10.7 LES CONCLUSIONS

Deux résultats majeurs se sont dégagés de ces séances de tests :

- Les constructions du langage permettent de programmer et d'expérimenter une multitude de mécanismes temporels qui augmentent l'expressivité des processus générés.
- Les problèmes rencontrés et les solutions *ad hoc* qui ont été proposées ont montré l'intérêt pour le développement d'outils de plus haut-niveau facilitant la spécification de concepts musicaux tels que le rubato, les changements continus du tempo, la relation lead/suiveur, ou la latence.

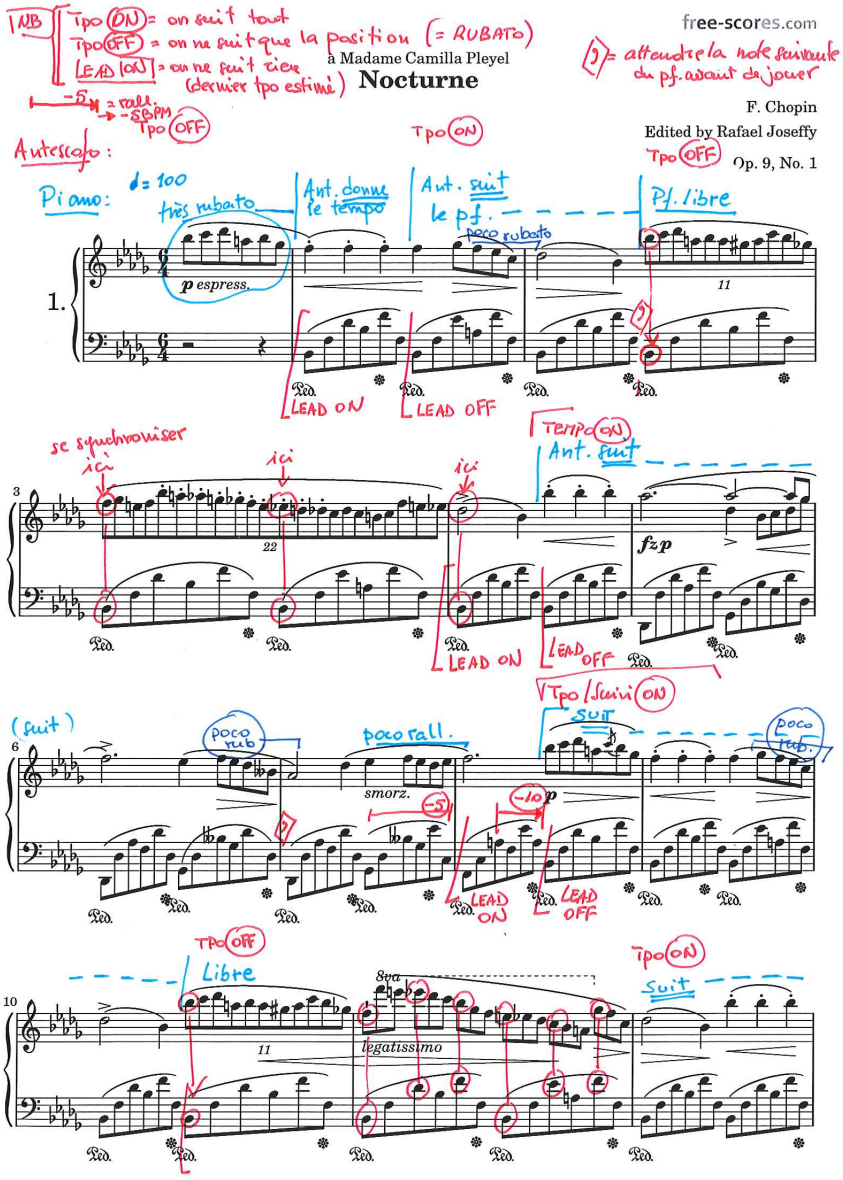



FIGURE 41: Extrait du Nocturne nº 1 de Frédéric Chopin avec les annotations de Marco Stroppa. Les différents mécanismes temporel du langage permettent de transposer ces annotations pour assurer un comportement expressif de l'accompagnement.

# 11 ACCOMPAGNEMENT AUTOMATIQUE

On définit l'accompagnement automatique par la synchronisation d'un flux MIDI ou audio avec le jeu d'un ou plusieurs musiciens. Les logiciels de suivi de partition sont en règle générale uniquement destinés à ce type d'application. Bien que le langage d'*Antescofo* ait été conçu pour en faire davantage, nous nous sommes intéressés à programmer des processus d'accompagnement automatique. Des collaborations avec différents musiciens et l'Orchestre de Paris ont permis de tester nos différents développements qui pourraient également aboutir à des applications pédagogiques.

## 11.1 ACCOMPAGNEMENT D'UN FLUX MIDI

Les premières réalisations d'applications pour l'accompagnement consistaient à synchroniser un groupe d'actions pour contrôler le séquencement de notes MIDI.

Un convertisseur permet de transformer un fichier MIDI en partition *Antescofo* : la première voix est convertie en la partie du musicien que le système doit suivre et les autres sont transformées en groupes s'exécutant en parallèle.

La stratégie de synchronisation `tight` était une première alternative à la stratégie `loose` (section 6.4). Elle simplifiait la conversion en évitant d'associer chaque action à l'événement correspondant et elle maintenait une parfaite synchronisation avec le musicien mais aussi entre les groupes :

```
NOTE 60 1.0
   group voix1 @tight
   {
      action1_1
      1/4 action1_2
         ...
   }
   group voix2 @tight
   {
      action2_1
      1/2 action2_2
         ...
   }
NOTE 54 1/4
   ...
```

Cette stratégie convient en situation de simulation, c'est-à-dire lorsque la partition est exécutée à partir d'un enregistrement audio. En re-

vanche en situation réelle, où un musicien joue sa partie et écoute l'accompagnement, la musique a une tendance générale à ralentir. Cela est dû au fait que l'accompagnement est toujours joué en réaction aux événements du musicien alors que le musicien attend un minimum d'anticipation de la part de la machine, ce qui provoque le ralentissement.

Pour remédier à ce problème, la stratégie `@ante` a été développée afin que l'accompagnement ne soit pas toujours en attente des événements du musicien (cf. section 6.4.1).

Cette stratégie est satisfaisante pour résoudre le problème de la synchronisation d'événements discrets mais n'offre pas de solutions immédiates pour la coordination d'un flux continu.

## 11.2 PROGRAMMER LA COORDINATION D'UN FLUX CONTINU

À ce stade, les programmes *Antescofo* des pièces précédentes qui séquençaient des fichiers audio, se contentaient de lancer les fichiers à la détection d'un événement en les jouant en l'état ou en modifiant la vitesse de lecture en fonction de l'estimation de tempo en temps réel. Afin d'affiner l'interaction entre le musicien et l'électronique dans ce genre de situation, nous avons écrit un programme pour synchroniser un flux audio de manière continu.

Notre objectif était de programmer, en utilisant le langage, un mécanisme permettant d'adapter la vitesse de lecture d'un vocodeur de phase pour assurer une coordination précise et continue d'un fichier audio.

La partition est alors organisée de la manière suivante :

- Les dates associées à la pulsation du fichier audio sont stockées dans un tableau.
- Un `whenever` permet d'envoyer une nouvelle valeur de la vitesse à chaque événement détecté.
- Cette vitesse est calculée en fonction de la position courante du fichier, de la position et du tempo courant du musicien de façon à viser une synchronisation au bout d'un certain nombre de pulsations.
- La vitesse est réajustée au moment de la synchronisation et divers mécanismes permettent d'éviter des changements trop brusques.

Ce programme a été utilisé pour des démonstrations d'*Antescofo* notamment à la conférence *ACM CHI 2013*, ou pour les *Objets de la nouvelle France industrielle*.

Ce code est cependant un peu *lourd* et donc difficile à adapter à différentes situations pour un utilisateur qui n'est pas un expert.

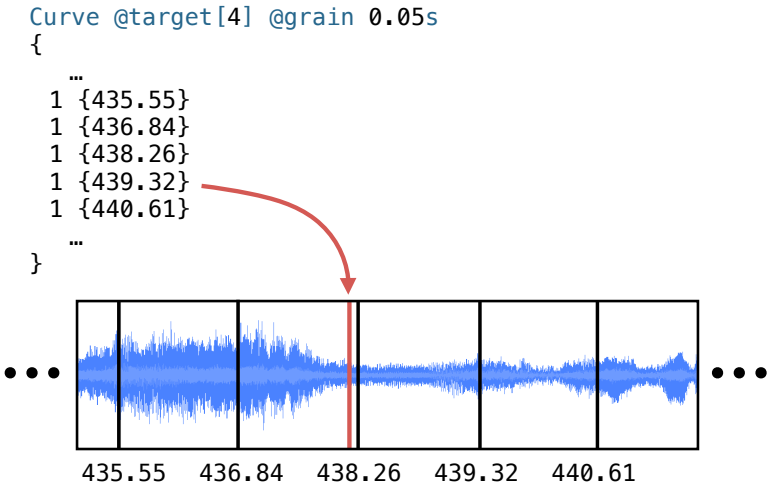



FIGURE 42: La curve permet d'envoyer au vocodeur de phase, la position à lire dans le buffer audio.

## 11.3 UTILISATION DES SYNCHRONISATIONS ANTICIPATIVES

Cela nous a amené à intégrer des stratégies de synchronisations anticipatives avec cibles statiques ou dynamiques (cf. section 6.4.2). Ces stratégies ont permis de simplifier la mise en oeuvre de pièces pour l'accompagnement automatique.

En associant une telle stratégie de synchronisation à un bloc d'actions, la position du bloc n'a plus besoin d'être gérée explicitement dans le programme. La relation entre le temps absolu du fichier audio et le temps relatif de la partition est gérée à l'aide d'une `curve` (cf. Figure 42). Chaque palier correspond à la date (en secondes) de la pulsation du fichier audio. La `curve` interpole ainsi la position du fichier audio qui est envoyée au vocodeur de phase.

L'utilisation de ces stratégies anticipatives est également adaptée pour contrôler des actions discrètes, comme celles correspondant à un flux MIDI, le lancement de fichiers audio courts, etc...

Ces stratégies ont été utilisées pour réaliser un accompagnement automatique pour plusieurs concertos pour piano (Mozart, Ravel, Liszt), dans le cadre de collaboration avec l'Orchestre de Paris et avec les pianistes Florent Boffard, Bertrand Chamayou et Jacques Comby.

# 12

# CANON DE TEMPOS EN TEMPS RÉEL

La présentation de ces travaux est extraite de [TE14].

Electroacoustic performance has long relied on the method of "front-end" synchronization, allowing live players to interact with electronics by triggering events in real time, with processes timed to unfold from these interspersed cues. With a reliable score-following tool coupled with a domain specific language, different strategies of synchronization have become feasible, notably the possibility of aligning at the end of a musical phrase.

Using the tempo canon principles of Conlon Nancarrow ([Gan95],[Thooo]) as a model, voices at related but independent speeds "catch up" at a predetermined convergence point (Fig 43). Unlike Nancarrow's rigid grids however, a degree of rhythmic flexibility and vitality becomes possible with the dynamic score-following tool *Antescofo*, whose programming structure allows high-level macro controls that adapt to real-time tracking data, adapting the playback speed of the canons accordingly to accompany the inevitable fluctuations of a live performance. Electronic events are timed no longer from a given starting point, but in relation to an approaching moment whose distance is continually predicted and recalculated.

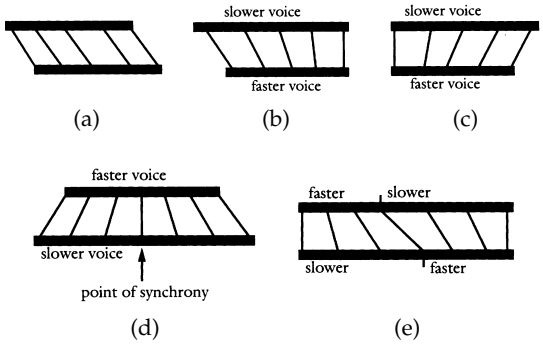



FIGURE 43: (43a) conventional canon. (43b) converging canon. (43c) diverging canon. (43d) converging diverging canon. (43e) diverging converging canon.

REALIZATION Sketches written by Christopher Trapani and played by the clarinetist Jérôme Comte from the *Ensemble intercontemporain* formed the basis of a six-month study, culminating with a 3-minute final sketch viewable here [vid13]. In this section we will detail the various challenges addressed during this collaboration.

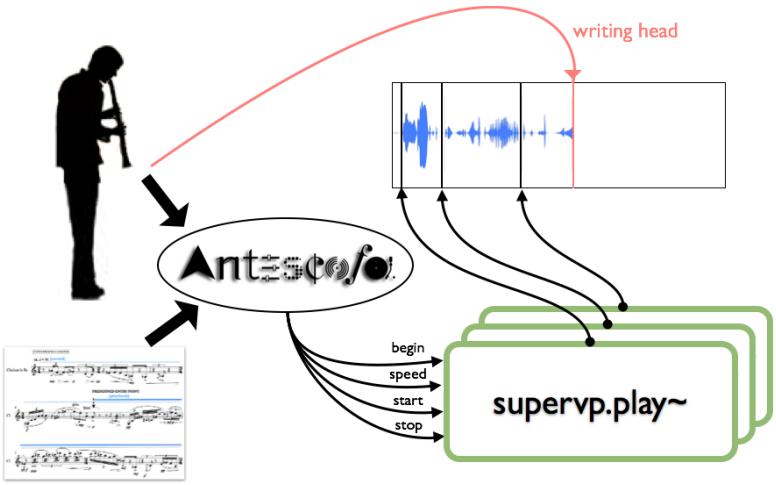



FIGURE 44: *Antescofo* follows the clarinetist's performance which is recorded in parallel into a buffer. Phase Vocoder objects read this buffer at specific positions and speeds controlled by *Antescofo* as outlined in the score.

SET-UP These sketches were realized in the Max/MSP environment. The live performer is recorded into an audio buffer, while a Phase Vocoder object (*SuperVP* [Roe03], [RR05]) reads this buffer at specific positions and speeds. The *Antescofo* program calculates the correct position and speed values according to incoming real-time data, communicating with the Phase Vocoder.

The challenge of creating a dynamic canon lies in building, estimating, and updating in real time the relation between the temporality of the score (in beats) and absolute time. All parameters that control the unfolding of the canon depend on this relationship. The *Antescofo* language allows the composer to manage this kind of control in a precise fashion.

DESCRIPTION OF REAL-TIME CANONS A canon can be defined as the superimposition of a phrase upon a version of itself. In our work, the live musician provides the first voice of the canon, whereas the electronics provide a manipulated second voice in reply. A two-voice canon can thus be characterized as a symbiotic phrase with two components :

- the canonic reply played back from a buffer (the line segment [start,end] in the figures 45),
- the phrase performed live by the musician onto which the canonic reply is superimposed (the line segment [start',sync] in the figures 45).

Keeping in mind that the playback position can never overtake that of the live recording for obvious causal reasons, different types of real-time canons can be defined (Fig 45).

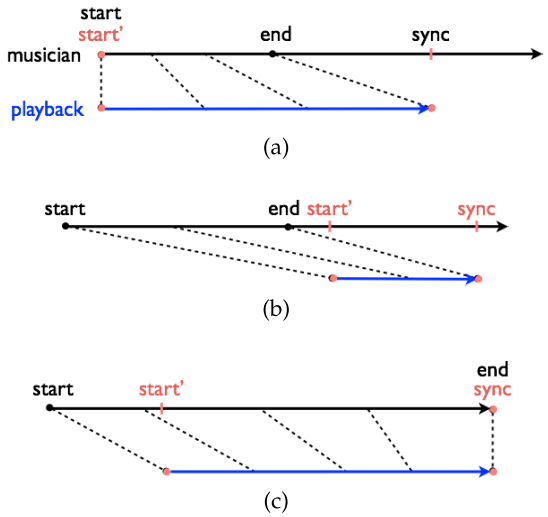



FIGURE 45: Different types of real-time canons

The temporal data pertaining to each canon is specified in the score. These specifications are virtual and relative to the tempo so that during performance, these values can be translated into a real and absolute scale.

The canon (45a) starts at the same time as the musician but unfolds at a slower speed. The canon (45b) starts after the musician has played the end of a phrase. This corresponds to a superposition of two independent voices. The canon (45c) begins after the live voice at a slower tempo that will gradually increase until catching up the musician at a pre-defined convergence point.

CANON SPEED CALCULATION In order to determine the precise speed for each canon in the score, the *Antescofo* program uses real-time tempo calculations as well as temporal estimations of upcoming events. The tempo of the musician as estimated by the listening machine of *Antescofo* is the key to calculating these estimates of future events. Thus, the speed values are re-estimated throughout the entire duration of the canon as soon as new information concerning the position or the tempo of the musician arrives.

Figure 46 represents the different relationships which can be found in the three types of canons mentioned above.

Let $t$ the current position of the musician in the absolute scale ; the computation of the speed follow this equation :

$$canonSpeed = \frac{end - t'}{sync - t} \tag{7}$$

where $sync - t$ is the estimation of the time before the synchronization point and $end - t'$ is the estimation of the time that remains to be played by the canon until the synchronization point.

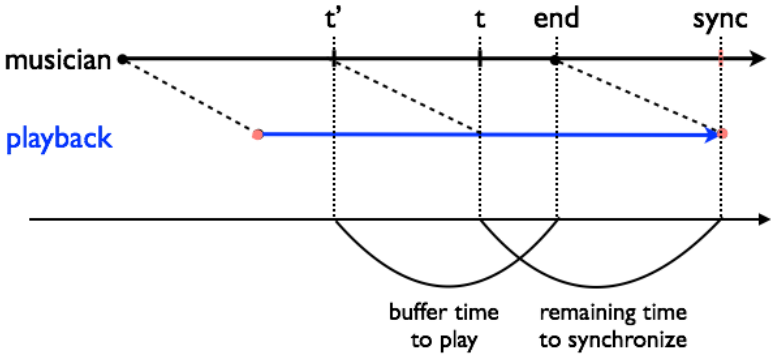



FIGURE 46: The diagram represents the different values to be considered for the speed calculation. At the current time t, the dates end (the canon end) and sync (the final synchronization point) are estimated according to the tempo value. The t' value corresponds to the time at which the live musician played the current position of the buffer.

This is the main piece of code in the score that computes the speed of the canon each time the tempo is updated :

```
whenever($RT_TEMPO)
{
 $t':=$t'+((@date($t')-$NOW)*$speed)

 $end:=$NOW+($end_pos-$RNOW)*60/$RT_TEMPO
 $sync:=$NOW+($sync_pos-$RNOW)*60/$RT_TEMPO

 $speed := ($end-$t')/($sync-$NOW)
 ph_voc speed $speed

}until ($RNOW ⩾ $end_pos)
```

`$NOW`, `$RNOW`, and `$RT_TEMPO` are internal variables that represent, respectively, the current time in seconds, the current position of the musician in beats, and the current value of the tempo.

The `whenever` statement allows actions to be launched when a given condition is verified. Each time the variables corresponding to this condition are updated, the predicate is re-evaluated. Here the body of the `whenever` is executed each time the variable `$RT_TEMPO` changes. This process continues until the live musician has reached the end of the canon (`$RNOW` ⩾ `$end_pos`).

`$t'` represents the audio time-chunk in the buffer already played ; `@date(t')` is the date of the last update of `$t'` variable. The tempo estimation is used to predict the dates of `$end` and `$sync`. `$speed` represents the speed of the playback and is used as the control parameter of the phase vocoder.

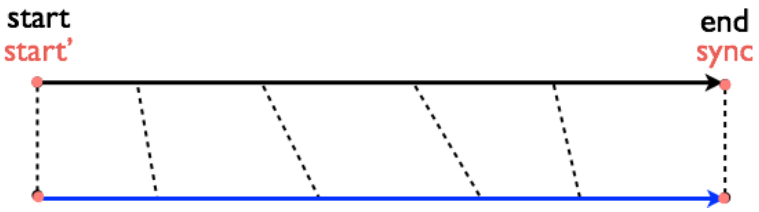



FIGURE 47: Real-Time Canons with speed curve constraint

REAL-TIME CANONS WITH SPEED CURVE CONSTRAINTS Traditionally, canons are defined by a constant speed ratio. Phase vocoder playback however offers the possibility of dynamic tempo flexibility following a predefined pattern. These shifts in speed can be easily stored as a function graph, with care taken to avoid increased playback speeds that would cause the playback buffer to overtake the recording. Equally applicable to prolongation and converging canons, these tempo maps provide a function transfer, an intermediary multiple that imposes a second set of speed relationships onto the canonic reply.

This strategy is also used to create a specific style of canon (type e in Fig. 43) where two voices cover the same length, both beginning and ending together, but with dynamic changes of speed within this frame. Nancarrow makes frequent use of this effect, switching the speeds between a faster and a slower voice at the exact midpoint of the phrase for a flawless alignment. In our example, the tempo shifts dynamically. From an initial shared starting point, the playback voice unfolds at a slower speed than the live player's, giving the impression of a prolongation canon with a malleable tempo. Later in the canon, phrases in playback are repeated faster than their live renditions, with the tempo again continually calibrated against the approaching convergence point.

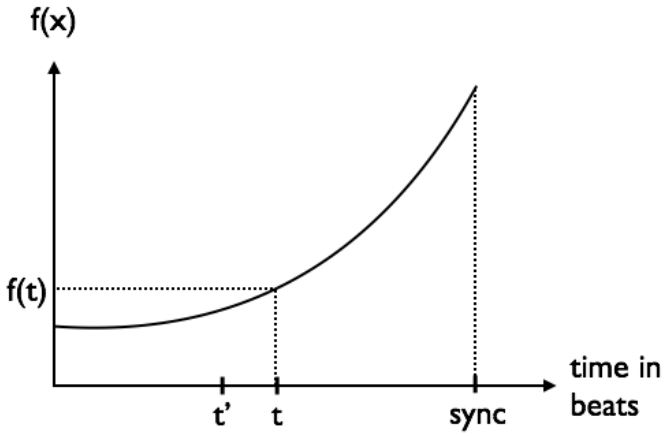



FIGURE 48: Example of a curve specified by the composer to constraint the speed of the canon.

The figure 48 shows a possible curve to constraint the speed of the canon. If $t$ is the relative position of the musician in the score, the sent speed is :

$$canonSpeed = f(t) * \frac{sync - t'}{\int_t^{sync} f(x)\,dx} \tag{8}$$

where f(t) is the current value of the curve, $sync - t'$ is the time in beats that remains to be played by the canon until the synchronization point and $\int_t^{sync} f(x)\,dx$ the integral of the curve from the current instant to the synchronization point.

In practice, the speed calculation is updated every 0.05 seconds.

METADATA INTERACTIONS Attack recognition data gathered in real time by *Antescofo* also serves as a tool for storing and recalling the positions of various events. As the live voice is followed, the timing in ms of each detected event is stored as a marker in the buffer. Though playback speed is continually in flux, the positions of these markers remain constant and function as signposts for various meta-processes.

As one basic example, these markers can be read as multiple entry points, so that canonic replies can begin not only at the start of the recorded buffer, but at any recognized event within the musical phrase. They may also serve as signposts for dynamic transposition, sending predetermined transposition values to the phase vocoder when the playback voice reaches the matching phrase.

Markers are also valuable for managing data outside of the signal realm. Using the onset and pitch data for each note in an Antescofo score, it is possible to keep track of the current pitch of each canonic voice, even as the playback rate changes. In our example, a process of adding and subtracting the frequencies of active pitches is established, with the results displayed in real time using the Bach interface [AG12]. The resulting pitches are used to pilot the transposition of concatenative synthesis in CataRT [SBVB06], producing a constantly shifting halo of samples whose harmonic content conforms to the sum and difference tones of the current pitches in the two canonic voices.

CONTROL AT THE CONVERGENCE POINT The most significant challenge of creating converging canons is how to treat the arrival point, since perfect synchrony between a live voice and a buffer is impossible. One short-term solution involves a last second crossfade in playback, from the buffer to the live voice. While this allows the final attack to be reliably aligned, the drawback is a loss of precision in the passage leading up to the convergence point. The playback speed is also systematically reset to 1. (parity) at the convergence point to prevent playback from overtaking the buffer's playhead.

A watchdog mechanism is executed to prevent playback from overtaking the buffer's playhead each time the speed value of the canon is updated :

```
whenever($RT_TEMPO)
{
 abort watchdog

 ...  canon speed computation ...

 group watchdog
 {
  $delay:=($NOW-$t')/($speed-1)
  $delay  ph_voc speed 1.0
 }
}until ($RNOW ⩾ $end_pos)
```

The watchdog mechanism is managed in the same body as the speed computation. The time needed for the position of the playhead (`$t'`) to catch up to the live player at the speed sent just before (`$speed`) is computed in the variable `$delay`. The message `ph_voc speed 1.0` is scheduled to be launched `$delay` seconds later. If the tempo is again updated, the `abort watchdog` command will cancel the message launch and re-schedule a new one.

The final moments of each converging canon involve increasingly perilous calculations. As the predicted arrival point approaches, tempo calculations have more immediate consequences, and small changes in the live player's timing can create drastic alterations in playback speed. With less time left to correct course, the danger of veering ahead of the live buffer's recording point increases.

A provisional solution has been the introduction of a braking mechanism which performs a secondary prediction. If the next event anticipated by *Antescofo* has not yet arrived after a given percentage of the allotted window (80% by default), the speed automatically slows along an exponential curve, so that playback gradually slows until the next predicted point of recalibration arrives. A prototype braking patch was created and tested during the work period, and should be integrated into the patch after further experimentation.

## 12.1 CONCLUSION

The success of this project demonstrates *Antescofo*'s capacity for sophisticated time-related processes, handling both live performance and real-time computing through a single streamlined engine. This could open the door to novel and imaginative reinventions in the realm of rhythm, extending the existing paradigm and encouraging new experiments with ancient musical devices in a wholly contemporary context. The musical examples explored thus far have been brief

and intuitive forays that only scratch the surface of the new possibilities offered by this approach.

If tempo canons can be realized in Antescofo with just a few lines of code, one can imagine a larger-scale work with several local-level tempo loops, converging and diverging in fractal-like patterns. The main barrier to composing such music today is the notation and execution of such a highly intricate level of rhythmic interplay, but a tool that generates canonic responses with real-time adaptive capabilities could lead to exciting new forays in the realm of labyrinthine rhythmic layering and detailed canonic writing.

The most important breakthrough for Nancarrow — the primary factor that paved the way for his unprecedented experimentation with multiple layers of tempo and timing — was being allowed to work out his ideas in visual and mathematical terms, directly onto the piano roll. The outcome of this project could signal a similar breakthrough in the electroacoustic realm : a new tool for intricately-timed canonic writing that could prove to be useful for many composers, enabling them to succinctly and reliably describe complex numerical relationships without recourse to notation.

# 13

# SUPERPOSITIONS DE COUCHES TEMPORELLES HÉTÉROGÈNES DANS UNE PIÈCE DE MUSIQUE MIXTE

Les améliorations continues de la machine d'écoute d'*Antescofo* et le développement du langage, en interaction avec le compositeur, permettent d'exprimer des relations temporelles de plus en plus fines. Si la composition de processus complexes enrichit l'écriture de l'électronique, la possibilité d'exprimer des relations entre les processus ainsi que leurs liens avec l'interprète impliquent de nouvelles écritures du temps pour la musique mixte. Les apports du temps-réel deviennent donc bien compositionnels dans la mesure où ils permettent d'exprimer une variété importante de temps musicaux et un rapport plus fin entre l'interprète et l'électronique.

La compositrice Julia Blondeau utilise *Antescofo* pour organiser et réaliser ses pièces avec électronique et participe également au développement du logiciel. Sa recherche porte sur la notion de couche temporelle. Dans ce chapitre, nous présentons quelques uns des mécanismes mises en œuvre dans *Antescofo* pour réaliser cette notion, à travers l'exemple de *Tesla ou l'effet d'étrangeté* (2014). Cette pièce a été créée au conservatoire de Lyon en 2014. C'est une pièce pour alto, orchestre et électronique. Une réduction pour alors et électronique a été jouée au Festival Musica en septembre 2014. Cette pièce de 24 minutes correspond à un programme *Antescofo* de plus de 17000 lignes. Dans cette composition, Julia Blondeau utilise plusieurs types de couches temporelles réparties selon deux concepts principaux :

- Dans le premier, chaque couche temporelle est vue comme l'élément d'un ensemble de plus grande taille partageant une même temporalité.
- dans le second, les couches temporelles sont hétérogènes et indépendantes les unes des autres. L'interprète représente lui-même une de ces couches. Les moyens de lier les différentes couches entre elles sont variés ainsi que les manières de l'exprimer. L'utilisation de mécanismes dynamiques s'avère ici indispensable.

L'intégralité de la synthèse est générée en temps-réel en fonction du tempo du musicien. Des processus lancés par *Antescofo* envoient les paramètres nécessaires aux générateurs de synthèse (écrits en CSound). Dans la figure Figure 49, six couches de synthèse entourent l'interprète et évoluent en parallèle (modulations de timbre, de vitesse, de fréquence et d'amplitude). Nous sommes ici dans le premier concept de couche temporelle expliqué plus haut. En reprenant ses termes, les lignes de synthèses évoluent avec l'interprète et consti-

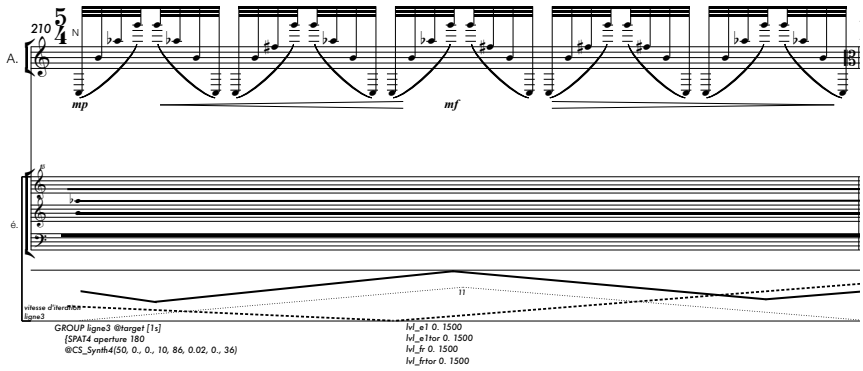



FIGURE 49: Extrait de la pièce *Tesla ou l'effet d'étrangeté* composée par Julia Blondeau. Les segments de droites représentent le déroulement temporel de processus électroniques se synchronisant avec la partie qui correspond à l'alto (portée du dessus).

tuent la « chair » d'un « squelette » incarné par l'interprète, et formant un seul et même temps.

L'écriture de points de convergence avec l'interprète peut se faire comme sur une partition « classique », permettant ainsi un travail de contrepoint réel entre les couches de synthèse et entre l'interprète et l'électronique. Au-delà de la question de la synchronisation, il s'agit ici d'écrire des points d'appui au sein d'une phrase, et d'exprimer des notions liées à la conduite et à la directionnalité de celle-ci, notions se trouvant à l'intersection de la composition et de l'interprétation. Chaque phrase peut avoir son propre tempo et l'évolution de ce tempo peut être dépendant ou non du musicien.

## 13.1 EXEMPLE DE SUPERPOSITION DE COUCHES INDÉPENDANTES

Dans l'extrait correspondant à la Figure 50, le musicien évolue dans un « territoire » divisé en quatre régions. Il doit choisir un parcours parmi des chemins possibles en traversant ces différentes régions. Certains processus électroniques sont associés à certaines cellules (les encadrés dans la Figure 50), tandis que d'autres couches temporelles se développent tout au long du parcours choisi.

Les processus correspondant à ces couches sont lancés lorsque le musicien rentre dans une région. Le tuilage entre régions se fait dynamiquement grâce à la spécification de commande `abort` et de handler de terminaison, définissant la manière de passer d'un processus à l'autre. Le tempo de chaque processus est indépendant de l'interprète, mais les processus demeurent néanmoins liés à son avancement dans la partition grâce aux stratégies de synchronisation utilisées. Certaines cellules jouées par le musicien permettent de « figer » le temps de ces processus, qui cessent, pendant une certaine durée,

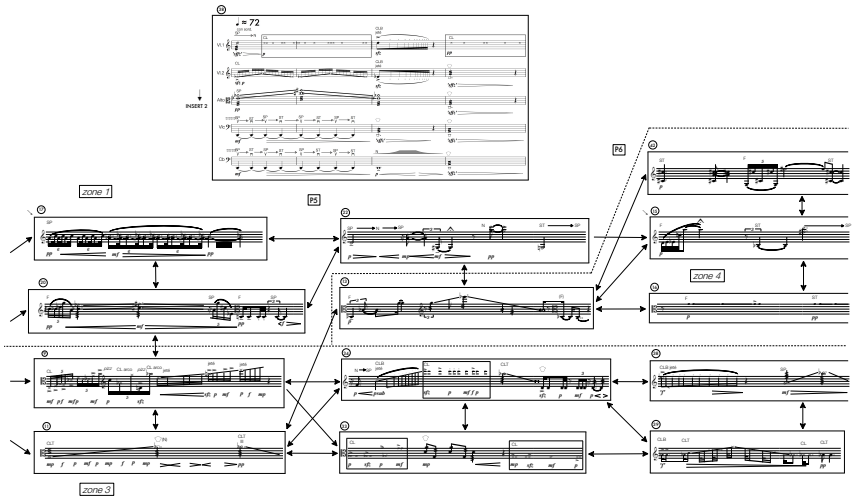



FIGURE 50: Extrait de *Tesla ou l'effet d'étrangeté* où le musicien choisit un parcours en suivant les flèches, allant d'une cellule à l'autre. Des processus électroniques sont associés à chacune des zones traversées, d'autres actions sont directement liées aux cellules.

d'évoluer. Il s'agit ici pour la compositrice de « composer des rapports dialectiques entre électronique et interprète, jamais unilatéraux mais toujours l'un à l'écoute de l'autre, dans leur propre indépendance ».

## 13.2 ORGANISATION HIÉRARCHIQUE D'UN PROCESSUS ÉLECTRONIQUE

Nous décrivons ici un processus électronique organisé selon des logiques temporelles à plusieurs niveaux (cf. Figure 51). Ce processus correspond à l'évolution d'une couche complexe de synthèse dont les variations continues sont contrôlées par un cycle qui module le spectre et les formes d'onde de la synthèse jusqu'à arriver à un « point critique » (l'étoile sur la Figure 51) déclenchant le mécanisme inverse. Cette évolution cyclique est contrôlée par le processus défini ci-dessous (ce code est directement extrait de la partition de concert) :

```
@proc_def ::RegionI ()
{
 $dur1RI := 16
 $tempoRegionI := $RT_TEMPO
 $modBattement := 0.0001
 $mfoAntes := 5.
 $batmax := 0.1

 ; COUCHEt1
 GROUP RegionI_CoucheT1 @target [2]
 ; longue acceleration jusqu'a mes52, puis deceleration jusqu'a
     mes67
```

```
{
   loop modFO $dur1RI
      {   abort interpBat
          abort mfoAntes
          curve interpBat @grain := 0.08s,
             @action := {    ASCOtoCS_SYNTH_L c ad2 ([ $nim($
                 modBattement) || $nim in $ModBat][1] )
                ASCOtoCS_SYNTH_L c ad3 ([ $nim($modBattement)
                    || $nim in $ModBat][2] )}
             { $modBattement
                {            { $modBattement } @type "quad"
                   ($dur1RI/2)   { $batmax } @type "quad_out"
                   ($dur1RI/2)   { 0.0001 }
                }
             }
          curve mfoAntes @grain := 0.08s,
             @action := ASCOtoCS_SYNTH_L c mfoAntes $mfoAntes
             { $mfoAntes
                {            { $mfoAntes }
                   ($dur1RI/2) { 8.85 }
                   ($dur1RI/2) { 5. }
                }
             }
          ($dur1RI) if ( $dur1RI >= 3.5 )
                       { $dur1RI := $dur1RI - 2 }
                       else { $dur1RI := 2 }
          if ( $batmax <= 0.8 )
                       { $batmax := $batmax + 0.025 }
                       else { $batmax := 0.8 }
      } until ("mes52" == $LAST_EVENT_LABEL)
    ...
```

Le processus `::RegionI` est altéré à des instants précis (encadré bleu de la Figure 51) : un certain type de tremolo joué par le musicien (quadruple cordes). Le tempo est alors ralenti, la spatialisation se fige et une fonction de modulation des harmoniques de la synthèse est mise en route. Cette modulation de la synthèse est plus forte à chaque apparition du type de tremolo reconnu.

```
whenever Trem4 ("Trem4_C3" == $LAST_EVENT_LABEL)
{
   $tempoRegionI := 20.
   $incremTrem4 := $incremTrem4 + 1
   ::SYNTH_Ant_tabAmpRand(($DURATION+2),0.03)

   $ampTrem := $ampGr + ($incremTrem4*0.015)
   curve ampGr @grain := 0.05s, @action := ASCOtoCS_SYNTH_L c
       kampAntes $NewAmpgr
      { $NewAmpgr
         {               { $ampGr } @type "cubic"
            ($DURATION/2) { $ampTrem } @type "cubic_out"
            (($DURATION/2)+2)   { $ampGr }
         }
```

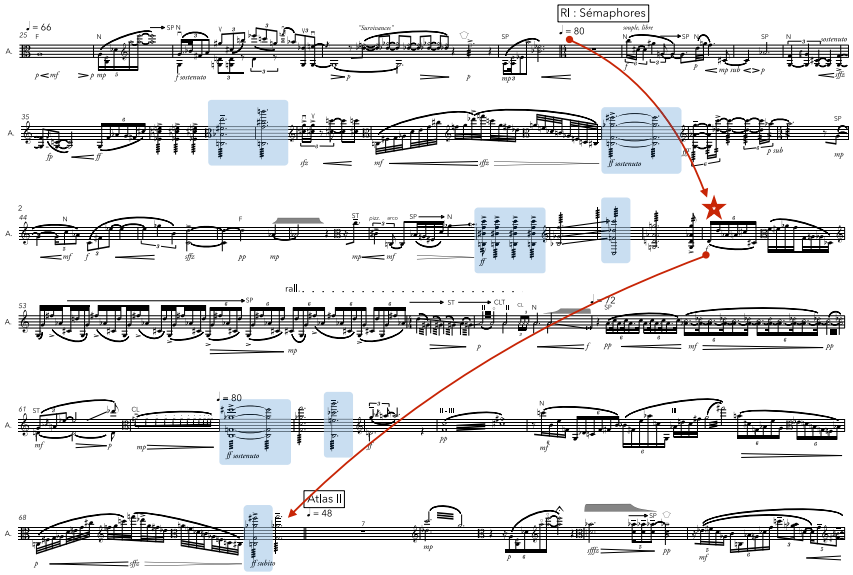



FIGURE 51: Extrait de *Tesla ou l'effet d'étrangeté*. Un processus électronique évolue de manière continue pendant toute la durée de cette séquence, tout en étant altérée de façon séquentielle lorsque certains événements du musicien sont détectés (cadres bleus).

```
        }
}
```

Nous voyons ici l'apport compositionnel du langage *Antescofo*. Dans la pratique « classique » de la musique mixte, la plupart des évènements sont déclenchés de manière séquentielle. La particularité de l'exemple que nous venons d'expliciter est que la partition *Antescofo* définit des processus qui évoluent à plusieurs échelles temporelles et selon des types d'évolutions et de déclenchements différents (évènementiels ou continus). Il y a d'une certaine manière une explicitation formelle de ce passage au sein même la partition, ce qui montre l'intérêt du langage d'un point de vue compositionnel et pas seulement technique. La compositrice a pu ainsi écrire dans sa partition *Antescofo* un processus qui aurait été écrit de façon totalement non explicite dans une partition « classique », par un enchainement de séquences dont la définition ne serait pas différenciée des autres types d'actions électroniques. Ici les processus sont définis à l'échelle à laquelle ils évoluent (dans ce cas précis, en parallèle à la partition). Les conditions logiques des `whenever` qui déclenchent le processus deviennent alors des sortes de « marqueurs formels » inscrits dans la partition et liés par ce moyen au temps de l'interprète.

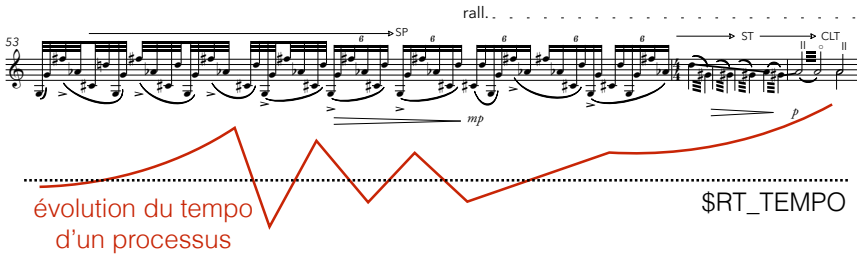



FIGURE 52: Extrait de *Tesla ou l'effet d'étrangeté*. L'évolution temporelle d'un processus électronique est décrit par une `curve` qui modifie le tempo du processus autour du tempo du musicien estimé en temps-réel.

## 13.3 TEMPO

Dans le passage de la Figure 52, la compositrice décrit un contrepoint entre l'alto et l'électronique dans lequel la partie électronique fluctue temporellement autour de la partie d'alto.

L'écriture est ici assez simple puisque les durées et rythmes sont exprimés de manière identique, en valeurs relatives et toutes égales. C'est la modulation du tempo qui rend ici le « geste malléable », tout en étant lié en permanence au tempo de l'interprète via la variable `$RT_TEMPO`. Pour les dynamiques, le même principe est employé grâce à l'utilisation d'une variable commune à toutes les notes (variable `$ampexplo`) envoyées au moteur de synthèse. On peut ainsi écrire directement une courbe comme on écrirait un crescendo sur une partition au lieu d'écrire une valeur d'amplitude pour chaque note. Le programme ci-dessous réalise à cette description :

```
Curve tempCouchT3 @grain := 1/12
{ $tempCouchT3
   {      { ($RT_TEMPO-5) } @type "cubic"
      2    { ($RT_TEMPO+40) }
      1/3   { ($RT_TEMPO-40) }
      1/2 { ($RT_TEMPO+30) }
      1/2 { ($RT_TEMPO-15) }
      1/2 { ($RT_TEMPO+20) }
      1/2 { ($RT_TEMPO-15) }
      3/2 { ($RT_TEMPO+20) } @type "cubic"
      7/3 { ($RT_TEMPO+60) }
   }
}
GROUP CoucheT3 @target {mes53,sync53,mes54,sync54_1,sync54_2,sync
    54_3,mes57}
      @tempo := $tempCouchT3
{
     ::SPAT_lissaj3("SPAT7",1.5,12,0)
     curve ampexplo @grain := 0.05s
        {$ampexplo
           {      { 0.08 } @type "cubic"
```

```
            2   { 0.19 } @type "cubic_out"
            2    { 0.09 }
            2    { 0.23 }
            2    { 0.09 }
            3    { 0.05 }
         }
      }

   ::ASCOtoCS_points("i33",1/8,$ampexplo,0.9,62)
   1/8 ::ASCOtoCS_points("i11",1/8,$ampexplo,0.6,87)
   1/8 ::ASCOtoCS_points("i11",1/8,$ampexplo,0.6,91)
   1/8 ::ASCOtoCS_points("i11",1/8,$ampexplo,0.6,67)
      ...
```

Les données rythmiques et dynamiques sont donc, d'une certaine manière, données de manière symbolique (grâce à la spécification par variable) et contrôlées de façon globale. Ce type d'écriture est extrêmement utile puisque très expressive (le concept est très lié à ce qui se passe à la fois sur la partition et à la manière dont un musicien interprète les symboles qui y sont inscrits). Toujours liée au tempo du musicien, la partie électronique s'adapte en permanence et ces facilités de « programmation » permettent alors une écriture fine, à l'échelle locale, de l'électronique.

## 13.4 MODES DE JEU

Lorsqu'un compositeur écrit pour un instrument, il a sous la main un arsenal de symboles pour indiquer à l'instrumentiste, ce qu'on définit comme des « modes de jeu ». Dans la musique électronique, on peut voir les modules de synthèse, et même de traitement, comme des instruments. Souvent ces instruments doivent être définis intégralement et donc impliquent un grand nombre de paramètres. Le langage d'*Antescofo* permet de réaliser des sortes de bibliothèques de « modes de jeu » d'un instrument de synthèse, propre au compositeur, et ainsi de pouvoir écrire, comme dans une partition classique, des notations symboliques comme la position d'un archet (*sul pont.*, *sul tasto*, etc.) sans avoir à chaque fois à en expliciter les paramètres. Chaque mode de jeu est structuré sous la forme d'un tableau. Les NIM permettent de passer d'un mode de jeu à l'autre par interpolation.

## 13.5 EXEMPLE D'UTILISATION DES CIBLES STATIQUES

Dans la Figure 53, les stratégies de synchronisation par cibles statiques permettent de spécifier des points pivots importants sur lesquels la spatialisation doit impérativement être synchronisée avec l'interprète. Des structures de type `curve` contrôlent la distance à partir du centre d'une source sonore dans un espace de diffusion.

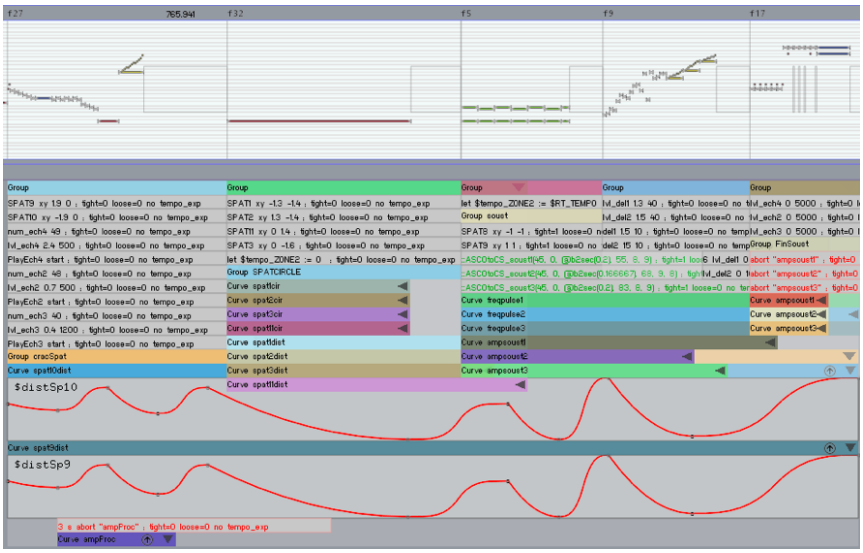



FIGURE 53: Visualisation de la partition augmentée de *Tesla ou l'effet d'étrangeté* dans l'interface graphique *Ascograph*. Dans cet extrait les paramètres de spatialisation sont contrôlés via des `curves` et synchronisés avec les événements du musicien avec une stratégie anticipative avec cible statique.

# 14

# GESTIONS DYNAMIQUES DE PROCESSUS POUR LE CONTRÔLE DE LA SYNTHÈSE DANS SUPERCOLLIDER

José-Miguel Fernandez est un compositeur et réalisateur en informatique musicale. En tant que RIM, il a notamment utilisé *Antescofo* pour la création de la pièce *Iki-no-Michi* de Ichiro Nodaïra (2012). Cette pièce utilise pour la première fois les structures de type `curve` permettant la spécification de paramètres continus qui s'adaptent au cours du temps de la performance.

En tant que compositeur, il a grandement contribué aux évolutions des mécanismes de communication avec l'environnement extérieur et au développement des structures de données et des fonctions.

## 14.1 COMBINER ANTESCOFO, MAX ET SUPERCOLLIDER

Dans sa pièce *Dispersion de trajectoires* (créée à Parme en 2015) pour saxophone et électronique, *Antescofo* est utilisé d'une manière originale en interaction avec les logiciels *Max* et *Supercollider* (cf. Figure 54). La partie du musicien est suivie par la machine d'écoute et permet non seulement de déclencher différentes séquences musicales électroniques mais aussi de coordonner un ensemble de processus créés et détruits dynamiquement à partir d'informations issues d'objets d'analyse audio.

Les traitements et la synthèse sonore temps-réel sont réalisés dans *SuperCollider*, particulièrement adapté à la création dynamique des processus de synthèse. L'analyse en temps réel du flux audio est réalisé dans *Max* avec les objets *Ircam Descriptors*.

## 14.2 ORGANISATION HIÉRARCHIQUE DES RÉACTIONS

Une grande partie de l'électronique générée pendant la performance est pensée de manière non-linéaire. Les processus sont organisés sous la forme de `whenever` qui sont à l'écoute pendant une durée délimitée dans la partition (structure de plus haut niveau). Ils sont activés lorsque des données d'analyse audio respectent des contraintes particulières. Ces données sont reçues à travers la commande *setvar* qui modifie directement la valeur de la variable concernée depuis l'environnement *Max*. Les descripteurs audio peuvent correspondre à des valeurs de centroïde spectral, de variations de spectre, d'amplitude, etc.

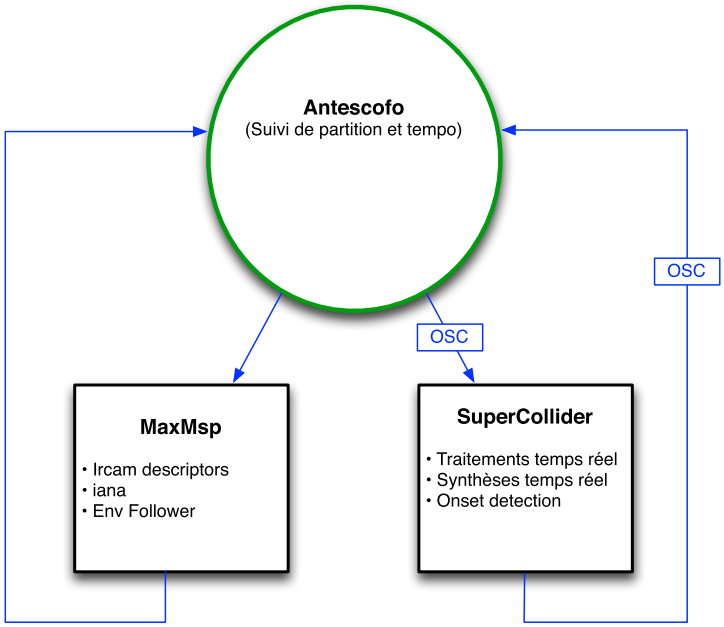



FIGURE 54: Architecture générale de *Dispersion de trajectoires* (2015) saxophone et électronique composé par José-Miguel Fernandez.

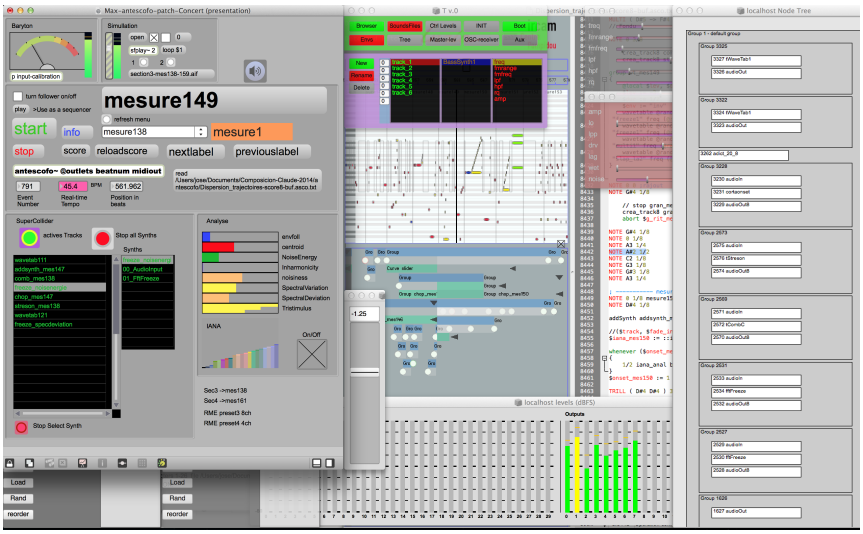



FIGURE 55: Capture d'écran d'écran de l'ordinateur exécutant les programmes de la pièce *Dispersion de trajectoires*

Les rectangles jaunes et mauves représentés dans la partition de la Figure 56 correspondent à ce type de processus. En voici un exemple de définition :

```
@proc_def ::spactral_var_trig_spec_freeze($track, $seuil,
                    $fade_in, $fade_out, $hp_prob, $amp, $prob)
@abort{fftfreezetrig $track off 1}
{
    @local $val, $last_val, $poly, $inc, $fade_in,
                $fade_out, $actif, $noterest
    $actif := 0
    $val := -1
    $last_val := -2
    $poly := 8
    $inc := 0

    fftfreezetrig $track (@vprob([$x | $x in 8], $hp_prob)) $fade
        _in $amp AudioInput input $saxIn #--FftFreeze bufft0
    fftfreezetrig $track amp -120

    0.2s $actif := 1
    whenever($actif && ($SpectVar==$SpectVar)){
       $val := ($SpectVar >= $seuil ? 0 : 1)
       if ($val != $last_val) {
          $last_val := $val
          $noterest := ((@rand(1.) < $prob)? 1: 0)
          if ($noterest == 1) {
             if ($val == 1) {
                fftfreezetrig $track pos ((@vprob([$x | $x in 8],
                     $hp_prob) * 2)/8)
                fftfreezetrig $track amp 0
                fftfreezetrig $track set 01_FftFreeze trig 1
             }else{
                fftfreezetrig $track set 01_FftFreeze trig 0
                fftfreezetrig $track amp -120
             }}}}}
```

L'appel du processus enclenche l'écoute du paramètre `$SpectVar`. Si cette variable dépasse un seuil passé en argument, alors une action sera déclenchée (un *gel spectral*) selon certaines probabilités.

Les canaux de sorties sont tirés aux hasard, mais les paramètres définissant les probabilités sont contrôlés au niveau de la partition linéaire pour élargir les possibilités au fur et à mesure de la performance.

Au passage, on peut noter un exemple de définition de tableau en extension, une fonctionnalité très utilisée pour manipuler des données vectorielles dans toute la pièce.

Le compositeur superpose ensuite plusieurs dizaines de processus similaires pour créer des polyphonies complexes.

Cet exemple est illustratif d'une pensée des événements à plusieurs niveaux. Le langage permet de définir les temps d'activation de processus à des dates bien précises, entre deux événements du musicien

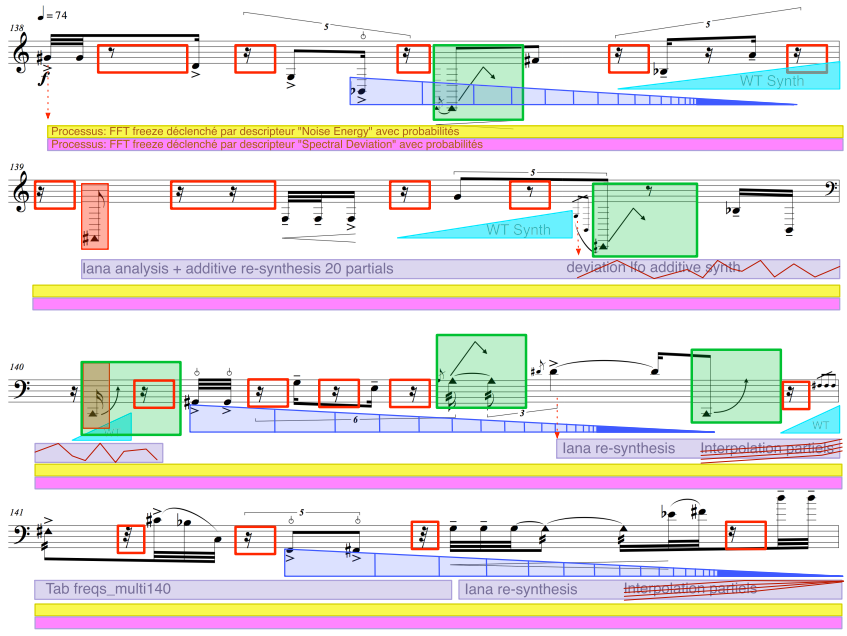



FIGURE 56: Extrait de la partition de José-Miguel *Dispersion de trajectoires*. Les

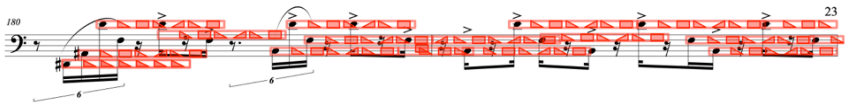


FIGURE 57: Superposition des couches de granulation synchronisées avec *Antescofo*.

par exemple. Plus localement, ces processus réagissent à des événements logiques et qui ne peuvent être déterminés avant la performance car ils dépendent de descripteurs audio calculés en temps réel.

## 14.3 SYNTHÈSE GRANULAIRE COORDONNÉE AU TEMPS DE LA PERFORMANCE

Un autre exemple tiré de cette pièce montre l'utilisation des spécificités du langage pour composer des processus électroniques de synthèse granulaire qui s'adaptent au temps de la performance (cf. Figure 57).

Des événements sont enregistrés pour constituer des grains qui sont ensuite rejoués plusieurs fois en suivant des rythmes rapides décrits par le compositeur. Chaque grain suit une enveloppe tirée au hasard parmi trois disponibles et est restitué dans un canal également tiré au hasard. Cela crée un effet de polyphonie complexe et aléatoire mais avec une temporalité bien définie.

## 14.4 CONCLUSION

Cette pièce repose de manière essentielle sur des fonctionnalités avancées du langage :

- la manipulation de tableaux représentant des paramètres de contrôle et la capacité à passer d'un jeu de paramètre à l'autre, de manière fluide et continu, en relation avec la temporalité du musicien.
- le lancement dynamique de processus, leur pilotage au cours du temps et la capacité à les perturber d'une manière non déterministe mais contrôlée via des mécanismes aléatoires complexes.
- l'intersection riche avec des outils extérieurs tant en entrée du sytème (descripteurs audio) qu'en sortie (contrôle de processus de synthèse dans Supercollider).

# 15

# SYNCHRONISATION DE PROCESSUS DE GÉNÉRATION AVEC IMPROTEK

*Antescofo* permet de spécifier des scénarios musicaux dont le contenu musical n'est pas connu à priori. Dans ces contextes de musiques improvisés ou semi-improvisés, la partition augmentée est un programme réactif qui va permettre de synchroniser des processus musicaux avec une ou plusieurs entrées. Nous détaillons l'utilisation du langage dans le logiciel d'improvisation *Improtek* pour gérer la temporalité de processus de génération et de réinjection. Un article propose l'association des logiciels *Importek* et *Antescofo* comme outil pour la planification de scénarios interactifs dans le cadre des musiques improvisées. Nous ne présenterons pas cette approche, le lecteur pourra se référer à [NECG14].

## 15.1 PROBLÈME MUSICAL

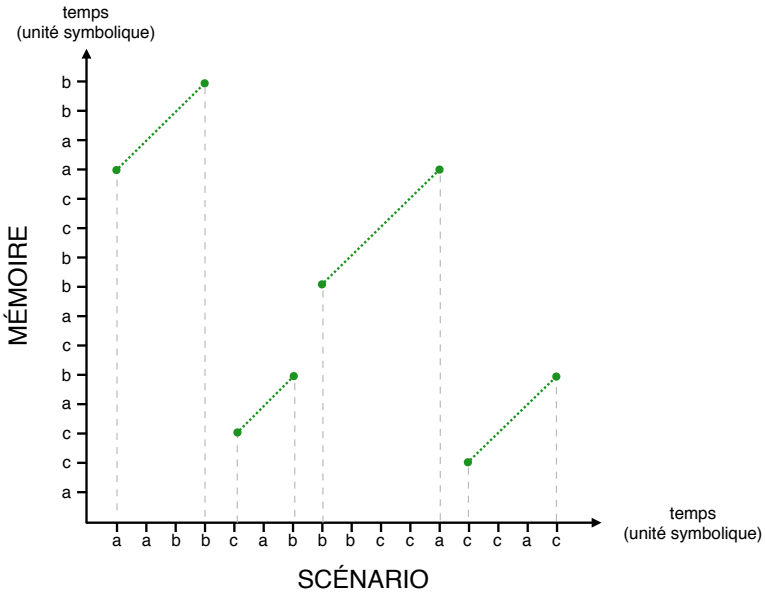
*Improtek* [NC15] est un outil permettant de naviguer dans une mémoire structurée en suivant un scénario donné. La mémoire et le scénario sont divisés en tranches qui correspondent à des pulsations. Les tranches sont étiquetées à partir d'un alphabet commun (par exemple des labels harmoniques).

Pendant la performance, l'environnement est capable de suivre le scénario prévu, de construire une mémoire à partir des données jouées par un musicien et de générer du matériau sonore en recombinant les tranches de différentes mémoires (y compris celles qui sont construites pendant la performance). Improviser revient donc ici à concaténer des segments de la mémoire qui satisfont les contraintes du scénario. Les détails sur les techniques de recombinaison sont disponibles dans l'article [NC15].

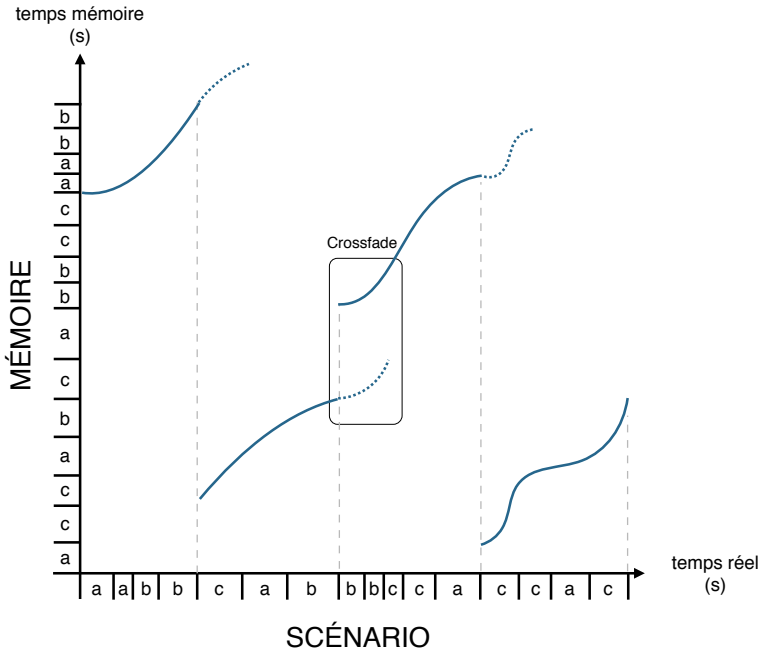
Nous nous concentrerons dans ce chapitre sur les mécanismes de synchronisation mis en place grâce au langage d'*Antescofo* pour coordonner l'écoute, l'annotation de la mémoire et la restitution des improvisations.

## 15.2 RÉALISATION

Toutes les entrées et les sorties audio sont gérées dans un patch Max. La mémoire musicale correspond à un buffer audio stocké dans Max. L'index des dates correspondant aux tranches (les pulsations) est construit dans un dictionnaire *Antescofo*.




(a) Modèle génératif : mapping symbolique.




(b) Restitution synchronisée : mapping temporel élastique.

FIGURE 58: Graphiques représentant le fonctionnement du logiciel Improtek. Un programme Open Music est utilisé pour re-combiner une mémoire symbolique selon un scénario symbolique. Un programme *Antescofo* synchronise ces résultats dans le temps de la performance en considérant les temps élastiques de la mémoire et de la performance en cours.

Les algorithmes de génération musicale guidée sont implémentés dans l'environnement OpenMusic. Pendant la performance, les appels à ces algorithmes sont effectués par une partition dynamique décrite dans le langage d'*Antescofo*. Les données symboliques générées par ces algorithmes indiquent les dates réelles de la mémoire à jouer pour chaque position du scénario. Ces données sont envoyés à *Antescofo* au fur et à mesure sous la forme de code *Antescofo* exécuté à la volée.

*Antescofo* est utilisé aussi bien en amont qu'en aval du système puisqu'il effectue les appels au modèle et séquence la restitution audio.

L'entrée de cette partition dynamique est une source de pulsation non métronomique. Le but est de jouer les phrases du modèle en s'adaptant aux variations de la source le plus musicalement possible. Cela est réalisé en alliant la gestion haut-niveau de la temporalité dans *Antescofo* et le time-stretch du vocodeur de phase (*superVP* en pratique). Les processus de restitution vont alors se synchroniser avec cette pulsation en utilisant les stratégies de synchronisation du langage.

Les voix de l'improvisation sont définies comme des instances d'un processus générique s'exécutant en parallèle.

Un processus écoute une source de pulsation à travers la mise à jour d'une variable `pos`. Celle-ci fournit une estimation du tempo et de la position courante dans l'improvisation créant un référentiel temporel sur lequel sont appliqués les stratégies de synchronisation. Ainsi la lecture du buffer s'adapte dynamiquement au temps de l'improvisation.

Pour gérer les discontinuités entre les tranches, le processus est constitué de structures imbriquées (un whenever, une loop et une curve) permettant d'envoyer les positions absolues aux vocodeur de phase. Tant que les tranches à lire dans le buffer sont continues, c'est la même instance de la boucle qui exécute les curves. En revanche si la prochaine tranche à lire est discontinue avec la précédente, alors un effet de *fondu enchaîné* est créé en superposant l'ancienne et une nouvelle curve de la nouvelle instance de la boucle.

Cette partition est un exemple caractéristique de l'utilisation du langage dynamique sans suivi de partition. Les propriétés dynamiques du langage permettent de spécifier des comportements temporels construits sur la relation d'événement et de durée pour des processus génériques.

# 16

# SYNCHRONISATION DE PROCESSUS DE GÉNÉRATION AVEC IMPROTEK

*Antescofo* permet de spécifier des scénarios musicaux dont le contenu musical n'est pas connu à priori. Dans ces contextes de musiques improvisés ou semi-improvisés, la partition augmentée est un programme réactif qui va permettre de synchroniser des processus musicaux avec une ou plusieurs entrées. Nous détaillons l'utilisation du langage dans le logiciel d'improvisation *Improtek* pour gérer la temporalité de processus de génération et de réinjection. Un article propose l'association des logiciels *Importek* et *Antescofo* comme outil pour la planification de scénarios interactifs dans le cadre des musiques improvisées. Nous ne présenterons pas cette approche, le lecteur pourra se référer à [NECG14].

## 16.1 PROBLÈME MUSICAL

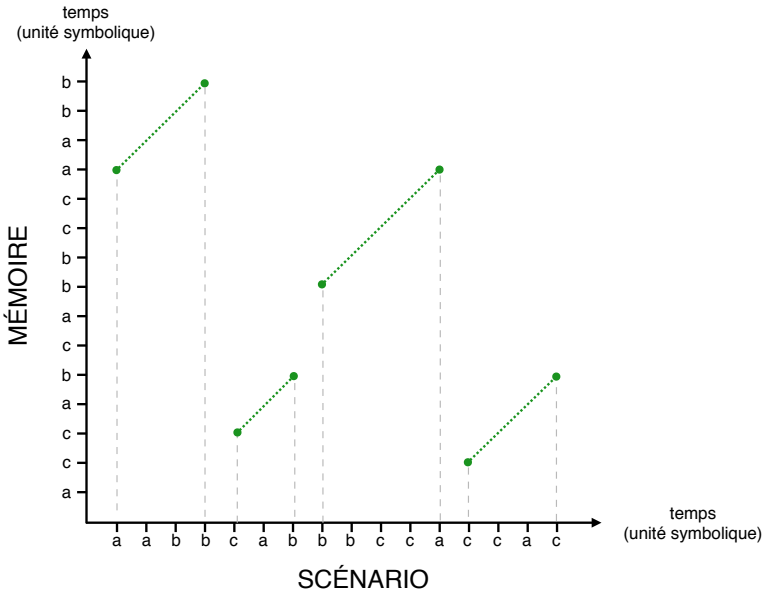
*Improtek* [NC15] est un outil permettant de naviguer dans une mémoire structurée en suivant un scénario donné. La mémoire et le scénario sont divisés en tranches qui correspondent à des pulsations. Les tranches sont étiquetées à partir d'un alphabet commun (par exemple des labels harmoniques).

Pendant la performance, l'environnement est capable de suivre le scénario prévu, de construire une mémoire à partir des données jouées par un musicien et de générer du matériau sonore en recombinant les tranches de différentes mémoires (y compris celles qui sont construites pendant la performance). Improviser revient donc ici à concaténer des segments de la mémoire qui satisfont les contraintes du scénario. Les détails sur les techniques de recombinaison sont disponibles dans l'article [NC15].

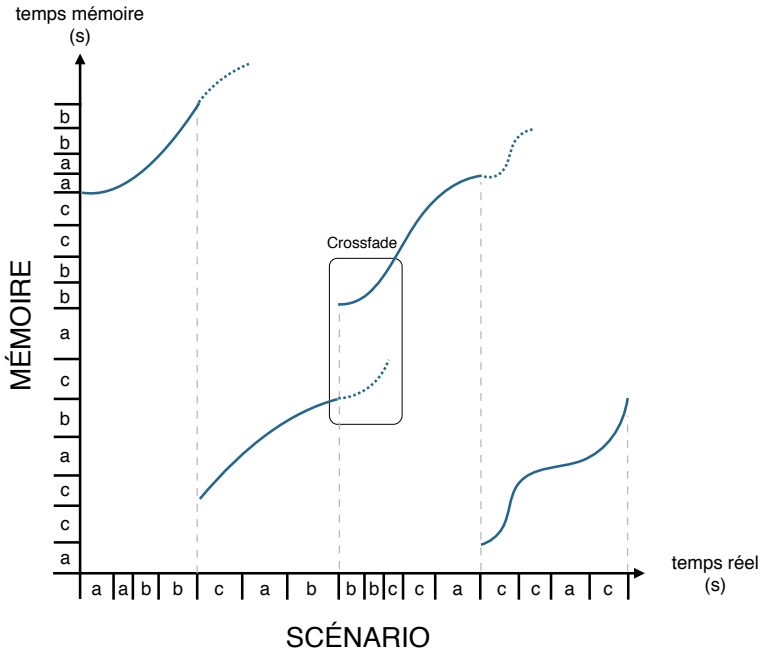
Nous nous concentrerons dans ce chapitre sur les mécanismes de synchronisation mis en place grâce au langage d'*Antescofo* pour coordonner l'écoute, l'annotation de la mémoire et la restitution des improvisations.

## 16.2 RÉALISATION

Toutes les entrées et les sorties audio sont gérées dans un patch Max. La mémoire musicale correspond à un buffer audio stocké dans Max. L'index des dates correspondant aux tranches (les pulsations) est construit dans un dictionnaire *Antescofo*.




(a) Modèle génératif : mapping symbolique.




(b) Restitution synchronisée : mapping temporel élastique.

FIGURE 59: Graphiques représentant le fonctionnement du logiciel Improtek. Un programme Open Music est utilisé pour re-combiner une mémoire symbolique selon un scénario symbolique. Un programme *Antescofo* synchronise ces résultats dans le temps de la performance en considérant les temps élastiques de la mémoire et de la performance en cours.

Les algorithmes de génération musicale guidée sont implémentés dans l'environnement OpenMusic. Pendant la performance, les appels à ces algorithmes sont effectués par une partition dynamique décrite dans le langage d'*Antescofo*. Les données symboliques générées par ces algorithmes indiquent les dates réelles de la mémoire à jouer pour chaque position du scénario. Ces données sont envoyés à *Antescofo* au fur et à mesure sous la forme de code *Antescofo* exécuté à la volée.

*Antescofo* est utilisé aussi bien en amont qu'en aval du système puisqu'il effectue les appels au modèle et séquence la restitution audio.

L'entrée de cette partition dynamique est une source de pulsation non métronomique. Le but est de jouer les phrases du modèle en s'adaptant aux variations de la source le plus musicalement possible. Cela est réalisé en alliant la gestion haut-niveau de la temporalité dans *Antescofo* et le time-stretch du vocodeur de phase (*superVP* en pratique). Les processus de restitution vont alors se synchroniser avec cette pulsation en utilisant les stratégies de synchronisation du langage.

Les voix de l'improvisation sont définies comme des instances d'un processus générique s'exécutant en parallèle.

Un processus écoute une source de pulsation à travers la mise à jour d'une variable `pos`. Celle-ci fournit une estimation du tempo et de la position courante dans l'improvisation créant un référentiel temporel sur lequel sont appliqués les stratégies de synchronisation. Ainsi la lecture du buffer s'adapte dynamiquement au temps de l'improvisation.

Pour gérer les discontinuités entre les tranches, le processus est constitué de structures imbriquées (un whenever, une loop et une curve) permettant d'envoyer les positions absolues aux vocodeur de phase. Tant que les tranches à lire dans le buffer sont continues, c'est la même instance de la boucle qui exécute les curves. En revanche si la prochaine tranche à lire est discontinue avec la précédente, alors un effet de *fondu enchaîné* est créé en superposant l'ancienne et une nouvelle curve de la nouvelle instance de la boucle.

Cette partition est un exemple caractéristique de l'utilisation du langage dynamique sans suivi de partition. Les propriétés dynamiques du langage permettent de spécifier des comportements temporels construits sur la relation d'événement et de durée pour des processus génériques.

# 17

# UTILISATION D'*ANTESCOFO* DANS UN GROUPE DE MUSIQUE ÉLECTRONIQUE

*Odei* est un groupe de musique électronique qui rassemble un vibraphoniste, un batteur et un musicien électronique. Leurs performances reposent sur l'utilisation de techniques d'improvisation jazz dans un contexte de musique électronique : les instrumentistes improvisent sur des séquences électroniques qui sont contrôlées et modulées par le musicien électronique. Ils utilisent *Antescofo* afin d'augmenter les interactions entre musiciens et machines pendant le concert.

Les deux instrumentistes partagent leur temps de jeu entre la construction en temps réel des phrases électroniques et l'improvisation sur leur instrument. Le musicien électronique manipule des effets et modifie la texture des sons joués par les différents synthétiseurs qui reçoivent les phrases dynamiques. Le but étant d'approfondir la dimension interprétative et humaine dans la musique électronique.

## 17.1 UNE PARTITION INTERACTIVE

Le programme *Antescofo* fonctionne comme un séquenceur dynamique qui reçoit les informations des contrôleurs (clavier, pads, interface dédiée), permettant de construire des structures qui sont ensuite déroulées dans le temps de la performance. Les contrôles sont répartis entre les trois instrumentistes qui agissent chacun sur des paramètres différents (rythmes, notes, etc.) des phrases électroniques envoyées aux synthétiseurs analogiques. L'organisation temporelle repose sur des boucles imbriquées (des structures de type `loop`) dont les périodes sont définies dynamiquement. Les informations reçues pendant un cycle façonnent les structures de ce cycle. Par exemple, à chaque fois qu'un événement des pads électroniques est reçu pendant le cycle, la structure de donnée correspondant au rythme de ce cycle est modifiée. Ce sont les durées entre les événements du même cycle qui définissent ce rythme. De nombreux contrôles permettent d'altérer les processus générés en modifiant le tempo, la quantification rythmique, l'ensemble des hauteurs, l'ordre hauteurs, etc.

Les structures temporelles et les structures de données d'*Antescofo* facilitent l'implémentation de ce type d'applications interactives qui cherchent précisément à allier la gestion des événements et des durées.

La musique jouée est répétitive et improvisée. Le système offre une grande liberté d'interprétation aux musiciens. *Odei* a eu l'occasion de

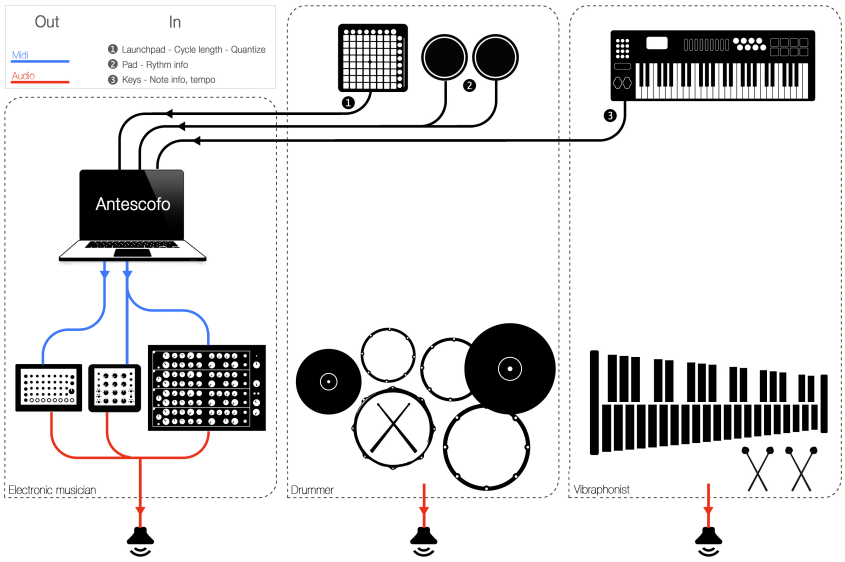



FIGURE 60: Dispositif de concert du groupe Odei. La partition *Antescofo* permet de construire des séquences MIDI à partir d'une combinaison des informations issues des pads électroniques et du clavier.

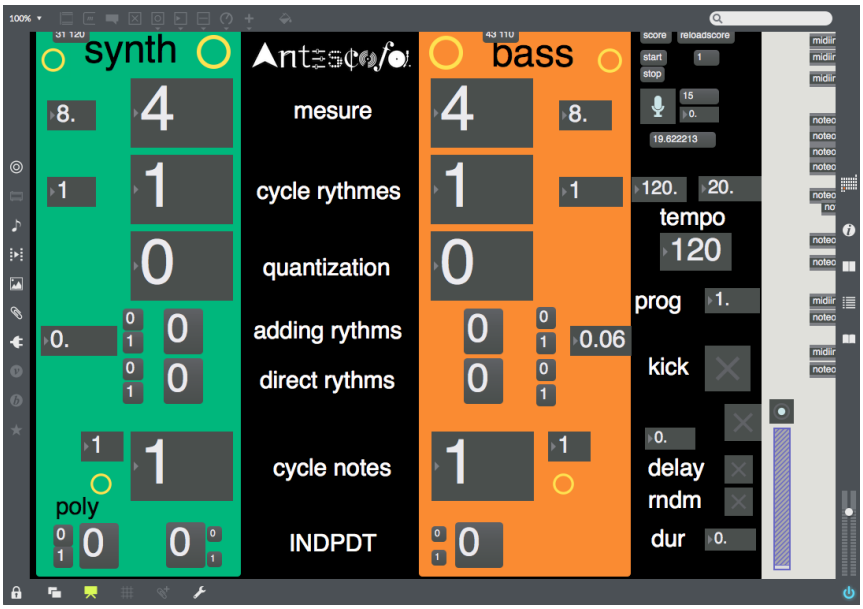



FIGURE 61: Interface indiquant l'état du système pendant la performance du groupe Odei.

se produire avec ce système dans de nombreuses salles de concert et festivals tels que le festival des *Nuits Sonores* à Lyon.

Quatrième partie

# CONCLUSION

# CONCLUSION ET PERSPECTIVES

*Antescofo* est un système cyber-physique où le musicien fait partie intégrante de la boucle de calcul, qui considère le temps musical comme un citoyen de première classe dans la sémantique de son langage. Ce langage à destination du compositeur permet de décrire la synchronisation musicale d'actions électroniques au jeu d'un musicien et d'imaginer et de construire des processus électroniques sophistiqués. Le système *Antescofo* est le résultat d'un couplage fort entre une machine d'écoute temps réel et un module réactif. Une partition *Antescofo* définit le comportement attendu de l'environnement (pour la machine d'écoute) et du système réactif. Le langage est pensé pour l'organisation dans le temps de processus musicaux hiérarchiques, concurrents et hétérogènes par rapport à un environnement musical fluctuant. Cette organisation est facilitée par la mise à disposition de constructions de haut niveau adaptées à la gestion dynamique des variables, des durées, des événements et de la coordination musicien-électronique.

Alors que le passage entre composition (prédéterminée) et interprétation musicale (non déterminée) se fait tout à fait naturellement chez les musiciens, peu de travaux ont essayé de modéliser et/ou d'abstraire les liens entre écriture et interprétation musicale dans la sémantique d'un langage de programmation. Mon travail dans *Antescofo* s'inscrit dans cette problématique. Au-delà du contexte musical, les questions soulevées et les réponses apportées pourront intéresser tout domaine dans lequel on doit faire face à la co-construction dans le temps d'un scénario prédéterminé.

Dans le reste de ce chapitre, nous allons passer en revue les lignes de force du projet, pour rappeler le travail réalisé et évoquer les perspectives qui s'ouvrent à présent.

## 17.2 CONCEPTS UTILES POUR LA MUSIQUE MIXTE

*Antescofo* a été utilisé lors de nombreux événements musicaux impliquant des ensembles tels que l'orchestre philharmonique de Berlin, l'orchestre philharmonique de Los Angeles ; et à travers des collaborations avec des compositeurs comme Pierre Boulez, Philippe Manoury et Marco Stroppa.

Malgré la jeunesse du langage, l'utilisation d'*Antescofo*, tant pour réaliser des pièces écrites avant l'apparition du langage, que pour de nouvelles créations, nous permet de tirer quelques conclusions quant aux concepts les plus utiles aux compositeurs :

- la possibilité de créer plusieurs systèmes de coordonnées temporelles à partir des événements musicaux et de la notion de tempo ;
- la possibilité de synchroniser ces systèmes de références temporelles ;
- la possibilité de créer dynamiquement un grand nombre de tâches ;
- la gestion des erreurs ;
- un couplage facile avec des processus extérieurs de natures différentes.

Nous pensons qu'*Antescofo* présente une alternative originale à la *tyrannie fertile de la timeline et du dataflow*. Par cette expression, nous voulons désigner deux représentations temporelles qui ont été utilisées de manière dominante pour la programmation des systèmes musicaux interactifs. Ces deux représentations, celle de la timeline séquentielle des séquenceurs classiques et celle du graphe à flot de données des logiciel comme *Max* ou *Open Music*, ont montré leur utilité, leur efficacité mais aussi des limites. Nous pensons que l'approche esquissée dans *Antescofo* représente une alternative, ou au moins une voie complémentaire qui permet d'échapper au caractère linéaire de la timeline mais aussi à la structure statique du patch.

Cette troisième voie, qui n'est pas antinomique des deux autres, doit encore faire ses preuves. Nous espérons qu'elle sera utile aux compositeurs et leur permettra d'étendre leur langage créatif.

## 17.3 LANGAGE RÉACTIF

Nous avons montré comment *Antescofo* s'inspire des langages réactifs utilisés dans les systèmes temps réel plus classiques, tout en se différenciant par des propriétés de dynamicité et d'élasticité du temps. Nous défendons l'idée que l'écriture et l'interprétation musicale fournissent un excellent champ d'applications pour la communauté des systèmes temps réel.

En effet, les systèmes musicaux interactifs sont moins contraints du point de vue de la vérification et de la validation que les systèmes temps réel critiques. Ils offrent donc plus d'opportunités d'exploration et nous sommes convaincus qu'ils sont exemplaires d'une nouvelle classe de systèmes cyber-physiques où les propriétés de sureté sont moins critiques que la capacité à assurer des relations complexes et changeantes.

Si *Antescofo* est ancré dans la famille des langages réactifs synchrones, la nécessité de gérer des durées, exprimées dans des repères temporels multiples, a amené à étendre le modèle événementiel standard. Il reste beaucoup de travail encore pour affiner la compréhension des mécanismes proposés dans cette thèse, et en particulier pour formaliser plus avant la notion de contexte temporel. Au-delà d'une sémantique de trace qui décrit l'effet des manipulations temporelles,

on peut espérer dégager à long terme une algèbre qui capture les propriétés essentielles des contextes (héritage, définition par synchronisation, etc.).

## 17.4 AUGMENTER L'EXPRESSIVITÉ DE L'ACCOMPAGNEMENT

Nous avons montré dans le chapitre 10 le besoin d'ajouter des constructions de haut-niveau pour la spécification de variations expressives du temps, qui permettent de mieux anticiper et donc de mieux coller au jeu du musicien. Une autre solution envisageable serait d'utiliser les répétitions et les enregistrements pour avoir une référence de ses variations. Cette technique n'est pas nouvelle, en témoigne l'article [VP85].

## 17.5 LA GESTION DES TEMPS MULTIPLES

Une problématique centrale de cette thèse est la modélisation et la gestion des temps multiples. Cette idée n'est pas propre à la musique mixte, mais elle demande dans ce dernier cas des représentations adaptées, pour analyser les temps de l'environnement et pour gérer les temps des processus électroniques. Les constructions offertes par *Antescofo* visent à faciliter ce genre de raisonnement. Le calcul dynamique des durées, des tempos et des mécanismes de synchronisation permettent de contrôler ou de contraindre indépendamment l'évolution de ces processus.

À court terme, nous envisageons d'introduire de nouveaux mécanismes allant dans ce sens, par exemple, pour spécifier les variations continues d'un tempo prises en compte directement dans les calculs des délais (pas d'échantillonnage) et en coordination avec le musicien. Ces variations pourraient être exprimées par des contraintes qualitatives, comme par exemple : commencer avec un tempo de 60, doubler ce tempo selon une une trajectoire polynomiale en 4 temps dans l'échelle du musicien avec une stratégie synchronisation `@tight`.

## 17.6 GESTION DE LA SIMULTANÉITÉ

La gestion de la simultanéité n'est pas encore aboutie dans le système *Antescofo*. Contrairement aux langages synchrones, deux actions ne peuvent jamais être considérées (même idéalement) comme simultanées. L'une sera toujours avant l'autre. Pour gérer l'ordre de ces actions, le système suit une règle assez naïve. La première action insérée dans la file d'attente est prioritaire. Cela peut donner des résultats contre-intuitifs pour l'utilisateur car cette règle ne respecte pas forcément l'ordre hiérarchique. Une solution dynamique d'attribution de propriété est en cours d'implémentation pour résoudre ce problème.

## 17.7 SYNTAXE ET INTERFACE GRAPHIQUE

Le système *Antescofo* s'insère dans un processus de création déjà établi où il interagit avec d'autres logiciels pour la musique. La syntaxe, pensée pour une prise en main rapide dans ce processus, reflète cette approche. Nous avons développé le langage de manière incrémentale en ajoutant les nouvelles fonctionnalités du système, au fur et à mesure des projets et collaborations artistiques. Des exigences de rétrocompatibilité nous ont parfois amené à introduire des notations pas toujours très intuitives.

Maintenant que le langage arrive à maturité avec une sémantique bien définie et qu'il prend de plus en plus en charge la gestion des processus électroniques des œuvres, il serait opportun de repenser une nouvelle syntaxe dans son ensemble.

D'autre part, le développement et l'usage de l'interface graphique *Ascograph* ont montré l'utilité d'une telle interface dans toutes les phases du processus compositionnel. La visualisation graphique des processus électroniques permet de mieux comprendre les relations entre les différents objets de la partition, et ce au moment de la composition, pendant les répétitions avec les musicien et pendant le concert. Le langage textuel peut alors se cacher derrière des syntaxes graphiques différentes et adaptées à un workflow particulier.

Cependant, des travaux sont encore nécessaires pour concevoir des représentations pour la visualisation des structures dynamiques et établir des relations fortes avec la sémantique du langage.

## 17.8 GÉNÉRALISER LA MACHINE D'ÉCOUTE

La machine d'écoute d'*Antescofo* fonctionne sur un principe assez simple (même si les méthodes utilisées peuvent être compliquées). Le spectre du signal d'entrée est comparé à un spectre généré correspondant aux notes de la partition. Des mécanismes facilitant la définition de spectre utilisateur, pouvant correspondre à des modes de jeu particuliers ou à des sons non harmoniques, pourraient permettre d'autres types de suivi.

Dans une autre direction, un travail à plus long terme est l'étude des méthodes de programmation bayésienne au sein du logiciel Antescofo, ce qui permettrait à l'utilisateur de décrire simplement avec des concepts de haut niveau des comportements dans un environnement ouvert et incertain.

## 17.9 VALIDATION

La question de la validation est délicate pour un système musical temps réel tel que celui d'*Antescofo*. D'abord parce qu'il adresse un problème spécifique sous un angle inédit ; les comparaisons avec les

autres systèmes du domaine ne sont donc pas immédiates. Ensuite, la syntaxe et l'expressivité du langage peuvent difficilement être validées autrement que par une collaboration étroite avec les utilisateurs (réalisateurs en informatique musicale, compositeurs et instrumentistes), pour répondre au mieux aux problématiques récurrentes de l'interaction musicien-machine. La singularité de l'*Ircam*, institut dans lequel ce projet a été développé, qui concentre dans le même lieu un laboratoire de recherche, des compositeurs ainsi qu'un service de création et de production, nous a permis de développer le langage en interaction constante avec ses utilisateurs finaux.

La validation formelle des programmes *Antescofo* excède le domaine de cette thèse, mais constitue de façon évidente une des directions de recherche majeure du projet et plusieurs travaux sont en cours [PSJ14, PJ15, BJMP13, FJ13].

Il semble difficile de pouvoir prouver une propriété quelconque sur les programmes les plus dynamiques, mais des programmes d'analyses statiques peuvent apporter beaucoup, et délimiter des fragments, correspondants à des ilôts de confiance ou au contraire à des zones dangereuses. De nouvelles techniques sont ici encore à inventer [Moro8].

## 17.10 PERFORMANCE

Concernant les performances du système, l'implémentation est pensée afin de répondre aux exigences qu'implique un concert de musique mixte : respecter les contraintes temporelles de plusieurs centaines de tâches s'exécutant en parallèle.

Différentes optimisations sont envisagées. Par exemple, l'échantillonnage de `curves` qui s'exécutent en parallèle peut ne pas être synchronisé et donc provoquer beaucoup plus de réveils que ce qui est réellement nécessaire. Des fonctionnalités pourraient être ajoutées pour faire en sorte que les actions se calent sur une grille temporelle de calculs. D'autre part, il serait également intéressant de faire en sorte que le grain d'interpolation puisse s'adapter en fonction des variations de la curve contrôlée : par exemple, envoyer l'action suivante lorsque la variation entre la valeur courante et la valeur précédente est supérieure à un certain seuil perceptif.

D'autres pistes vont être étudiées, telles que l'utilisation d'approches anticipatives et spéculatives par analyse des dépendances de données qui pourraient permettre d'optimiser l'ordonnancement des calculs. On peut également imaginer des modes de calcul dégradés qui réduiraient la charge lorsque le système détecte que les ressources deviennent trop limitées.

## 17.11 INTÉGRATION DU CALCUL AUDIO

À l'Ircam, certaines productions artistiques doivent gérer une grande quantité de processus informatiques pour la synthèse, l'analyse la spatialisation sonore, la vidéo, etc. et il arrive parfois que les RIM utilisent plusieurs instances du même logiciel pour éviter des surcharges de calculs.

Notre équipe explore actuellement différentes pistes pour introduire le calcul audio dans l'environnement *Antescofo* avec les librairies *Faust*, *CSound* et *Fluidsynth*. Les objectifs sont multiples :

- permettre aux utilisateurs de programmer l'ensemble des processus de contrôle et de synthèse correspondants au sein d'un même environnement ;
- contrôler symboliquement, précisément et de manière efficace les variations de paramètres de contrôle d'un signal audio ;
- modifier dynamiquement la chaîne de calcul audio ;
- tirer parti de la structure de la partition pour essayer d'optimiser ou d'anticiper les calculs audio et ainsi éviter les surcharges de calculs.

Par ailleurs, plusieurs propositions d'architectures temps réel émergent au pan international, comme CBS ou VBS, qui permettraient d'offrir des garanties de service même dans des cas très dynamiques. Ces nouvelles propositions doivent être prise en compte.

## 17.12 LA PARTITION AUGMENTÉE COMME NOUVEL OUTIL DE CRÉATION ET D'ANALYSE

De la même manière que les langages synchrones libèrent le programmeur des contraintes d'implémentation pour qu'il puisse se concentrer sur la sémantique de l'application, les outils de haut-niveau offerts par *Antescofo* visent à permettre au compositeur la retranscription directe de sa pensée, et à libérer son imagination. Il doit être plus facile pour lui de créer, de modifier et de supprimer les différentes couches temporelles qui constituent l'œuvre musicale sans avoir à se soucier des problèmes techniques de synchronisation.

La pensée de chaque compositeur lui est propre. Un des enjeux du langage, qui doit être validé sur le long terme, est de munir *Antescofo* de mécanismes suffisamment versatiles pour soutenir la pensée de chaque compositeur et pour développer ses idiosyncrasies propres. Cela veut dire aussi étendre le langage, avec des mécanismes permettant de développer des bibliothèques spécifiques, capturant le "vocabulaire" et les "design patterns" qui constituent le langage du compositeur. Il faut aller là plus loin que la notion de processus. La notion de pattern que nous avons développé dans [GE+14] va dans ce sens.

Par ailleurs, cela ne sera possible que si l'on dispose des outils permettant d'expliciter dans la partition le vocabulaire temporel d'une

pièce. On voit ici que *Antescofo* peut aider à une analyse des pièces. Associé à l'interface graphique *Ascograph*, le système peut se révéler une piste intéressante pour l'analyse musicale et un bon support pour la transmission et la préservation des œuvres.

## 17.13 VERS UN LANGAGE DE COORDINATION

Le modèle d'*Antescofo* est innovant par sa faculté à coordonner des processus électroniques avec le jeu d'un musicien. Nous envisageons d'élargir le principe de coordination à d'autres modules et applications (geste, vidéo, *SPAT*, *Ianix*, etc.).

Les recherches de l'équipe ISMM à l'Ircam (Interaction - Son, Musique, Mouvement) se focalisent sur l'interaction gestuelle avec des processus sonores ou musicaux, permettant par exemple de coupler le contrôle de la synthèse sonore aux mouvements d'un musicien ou d'un danseur, mesurés grâce à des capteurs de mouvement. En particulier, des méthodes de suivi de geste (avec un système comme Gesture Follower par exemple [BZS+10]) ont été développées, permettant d'effectuer en temps-réel la reconnaissance et l'alignement continu d'un geste avec un exemple référence, qui peut ensuite être traduit en paramètres sonores. Le suivi est alors continu et adaptable grâce à une phase d'apprentissage. Certaines productions de l'Ircam ont utilisé ce système en interaction avec le langage *Antescofo* mais d'une façon assez simpliste. Il serait intéressant d'étudier les modalités de couplage entre le suivi de geste ou autre module de mapping gestuel avec *Antescofo*, et l'intégration de modes d'interaction gestuelle dans l'environnement de programmation. Il faudrait pour cela étendre le langage existant par des constructions offrant aux compositeurs/utilisateurs une syntaxe et une sémantique simple, expressive et cohérente.

Cet exemple montre la nécessité de faciliter l'interopérabilité entre les différents modules du système (observation, reconnaissance, partition augmenté, moteur réactif).

Nous prévoyons également d'étudier le fonctionnement du système dans un contexte distribué. Comment écouter plusieurs musiciens ou plusieurs instances d'*Antescofo* en même temps ? Même si le système permet la gestion explicite de plusieurs référentiels en entrées et en sorties, la question de la déduction d'un référentiel à partir de plusieurs entrées reste ouverte. On peut également se poser la question du modèle de communication utilisé. Des langages tels que *HipHop* [BS14] pourraient nous donner quelques éléments de réponse.

## 17.14 VERS UNE AUTONOMISATION DU SYSTÈME

Nous avons vu comment le compositeur peut choisir la stratégie qui lui semble la plus adaptée en fonction du contexte musical et du processus électronique contrôlé.

De nombreux paramètres de ces stratégies peuvent être modifiés dynamiquement par des actions de la partition, mais il n'est pour l'instant pas possible de changer dynamiquement de stratégie. L'ajout de cette possibilité est en cours d'implémentation.

Serait-il alors possible (et intéressant) d'offrir la possibilité aux compositeurs de définir une méta-stratégie par un ensemble de règles, qui permettent au sytème de choisir la meilleure stratégie en fonction du contexte ? Et le système pourrait-il de manière autonome apprendre ces règles en *répétant* avec le musicien ? L'idée n'est alors plus que le compositeur programme la synchronisation mais que celle-ci émerge par apprentissage et évolution lors des répétitions, à partir de critères de haut-niveau spécifiés par le compositeur. Ces questions restent ouvertes.

# BIBLIOGRAPHIE


[ABC+06] G. Assayag, G. Bloch, M. Chemillier, A. Cont, and S. Dubnov. OMax brothers : a dynamic topology of agents for improvization learning. In *Proc. of the 1st ACM workshop on Audio and music computing multimedia*, pages 125–132, ACM, Santa Barbara, California, 2006.

[ACM02] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49(2) :172–206, 2002.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126 :183–235, April 1994.

[ada97] Ada 95 reference manual : language and standard libraries, 1997. International standard ISO/IEC 8652 :1995(E).

[AG12] Andrea Agostini and Daniele Ghisi. Bach : An environment for computer-aided composition in max. In *ICMC 2012 - International Computer Music Conference*, Ljubljana, Slovenia, September 2012. IRZU - the Institute for Sonic Arts Research.

[AG13] Andrea Agostini and Daniele Ghisi. Real-time computer-aided composition with bach. *Contemporary Music Review*, 32(1) :41–48, 2013.

[AHV93] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 592–601, New York, NY, USA, 1993. ACM.

[AK89] David P Anderson and Ron Kuivila. Continuous abstractions for discrete event languages. *Computer Music Journal*, pages 11–23, 1989.

[AK90] David P. Anderson and Ron Kuivila. A System for Computer Music Performance. *ACM Trans. Comput. Syst.*, 8(1) :56–82, February 1990.

[All83] J.F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11) :832–843, 1983.

[AMdS07] Charles André, Frédéric Mallet, and Robert de Simone. Modeling time(s). In Gregor Engels, Bill Opdyke, DouglasC. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 559–573. Springer, 2007.

[BAA09] Jean Bresson, Carlos Agon, and Gérard Assayag. Visual lisp/clos programming in openmusic. *Higher-Order and Symbolic Computation*, 22(1) :81–111, 2009.

[Bac78] John Backus. Can Programming Be Liberated from the Von Neumann Style ? : A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8) :613–641, August 1978.

[BB90] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 81–94, New York, NY, USA, 1990. ACM.

[BB91] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9) :1270–1282, 1991.

[BBCP12] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Non-standard semantics of hybrid systems modelers. *Journal of Computer and System Sciences*, 78(3) :877–910, 2012.

[BF14] Simon Bliudze and Sébastien Furic. An operational semantics for hybrid systems involving behavioral abstraction. In *Proceedings of the 10th International ModelicaConference*, number EPFL-CONF-199085, pages 693–706. Linköping University Electronic Press, Linköpings universitet, 2014.

[BG92] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language : Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2) :87–152, 1992.

[BG14] Jean Bresson and Jean-Louis Giavitto. A reactive extension of the openmusic visual programming language. *Journal of Visual Languages & Computing*, 25(4) :363–375, 2014.

[BGA06] Jean Bresson, Fabrice Guédy, and Gérard Assayag. Musique lab maquette : approche interactive des processus compositionnels pour la pédagogie musicale. *Revue des*

*Sciences et Technologies de l'Information et de la Communication pour l'Education et la Formation (STICEF)*, 13 :26–pages, 2006.

[BHN99] Elizabeth Bjarnason, Görel Hedin, and Klas Nilsson. Interactive language development for embedded systems, 1999.

[BJMP13] Guillaume Baudart, Florent Jacquemard, Louis Mandel, and Marc Pouzet. A synchronous embedding of antescofo, a domain-specific language for interactive mixed music. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–12. IEEE, 2013.

[BLM90] Jean-Pierre Banâtre and Daniel Le Métayer. The gamma model and its discipline of programming. *Science of Computer Programming*, 15(1) :55–77, 1990.

[Bon99] Bert Bongers. Exploring novel ways of interaction in musical performance. In *Proceedings of the 3rd conference on Creativity & cognition*, pages 76–81. ACM, 1999.

[Bou00] Richard Boulanger, editor. *The Csound Book : Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*. MIT Press, Cambridge, MA, USA, 2000.

[Bre00] Roberto Bresin. *Virtual Virtuosity Studies in Automatic Music Performance*. PhD thesis, KTH, Stockholm, 2000.

[Bre07] Jean Bresson. *Sound synthesis in computer-aided composition*. Theses, Université Pierre et Marie Curie - Paris VI, November 2007.

[BS01] Gérard Berry and Ellen Sentovich. Multiclock esterel. In *Correct Hardware Design and Verification Methods*, pages 110–125. Springer, 2001.

[BS14] Gérard Berry and Manuel Serrano. Hop and hiphop : Multitier web orchestration. In *Distributed Computing and Internet Technology*, pages 1–13. Springer, 2014.

[But97] Giorgio C. Buttazzo. *Hard real-time computing systems : predictable scheduling algorithms and applications*. The Kluwer international series in engineering and computer science. Kluwer Academic Publishers, Boston, 1997. 3e tirage 2000.

[BW01] A. Burns and A.J. Wellings. *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley Longman, 2001.

[BZS+10] Frédéric Bevilacqua, Bruno Zamborlin, Anthony Sypniewski, Norbert Schnell, Fabrice Guédy, and Nicolas Rasamimanana. Continuous realtime gesture following and recognition. In *Gesture in embodied communication and human-computer interaction*, pages 73–84. Springer, 2010.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4) :444–458, 1989.

[CGC14] Thomas Coffy, Jean-Louis Giavitto, and Arshia Cont. Ascograph : A user interface for sequencing and score following for interactive music. In *ICMC 2014-40th International Computer Music Conference*, 2014.

[Cha77] Joel Chadabe. Some reflections on the nature of the landscape within which computer music systems are designed. *Computer Music Journal*, pages 5–11, 1977.

[CLL+06] Adam Cataldo, Edward Lee, Xiaojun Liu, Eleftherios Matsikoudis, and Haiyang Zheng. A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems*, July 10-12 2006.

[CLRC05] C. Consel, F. Latry, L. Réveillère, and P. Cointe. A generative programming approach to developing dsl compilers. In R. Gluck and M. Lowry, editors, *Fourth International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 29–46, Tallinn, Estonia, September 2005. Springer-Verlag.

[Con63] Melvin E. Conway. Design of a Separable Transition-diagram Compiler. *Commun. ACM*, 6(7) :396–408, July 1963.

[Con08] Arshia Cont. Antescofo : Anticipatory Synchronization and Control of Interactive Parameters in Computer Music. In *Proceedings of International Computer Music Conference (ICMC)*, Belfast, Irlande du Nord, August 2008.

[Con10] Arshia Cont. A Coupled Duration-Focused Architecture for Real-Time Music-to-Score Alignment. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(6) :974–987, 2010.

[CP01] Pascal Cuoq and Marc Pouzet. Modular causality in a synchronous stream language. In *Programming Languages and Systems*, pages 237–251. Springer, 2001.

[CS06] Graham Coulter-Smith. Deconstructing installation art, 2006.

[Cut88] N. Cutland. *Nonstandard Analysis and Its Applications*. Cambridge Topics in Mineral Physics & Chemistry. Cambridge University Press, 1988.

[Dan84a] Roger B. Dannenberg. An On-Line Algorithm for Real-Time Accompaniment. In *Proceedings of International Computer Music Conference (ICMC)*, pages 193–198, Paris, France, 1984.

[Dan84b] Roger B. Dannenberg. Arctic : A functional language for real-time control. In *In 1984 ACM Symposium on LISP and Functional Programming*, pages 96–103, 1984.

[Dan97a] Roger B Dannenberg. Abstract time warping of compound events and signals. *Computer Music Journal*, pages 61–70, 1997.

[Dan97b] Roger B. Dannenberg. The implementation of nyquist, a sound synthesis language. *Computer Music J*, 1997.

[DB94] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 7th edition, 1994.

[DCA05] M. Desainte-Catherine and A. Allombert. Interactive Scores : A Model for Specifying Temporal Relations Between Interactive and Static Events. *Journal of New Music Research*, 34(4) :361–374, 2005.

[DLSW13] Alexandre Donze, Sophie Libkind, Sanjit A. Seshia, and David Wessel. Control improvisation with application to music. Technical Report UCB/EECS-2013-183, EECS Department, University of California, Berkeley, Nov 2013.

[DMK+06] Frederic Doucet, Massimiliano Menarini, Ingolf H Krüger, R Gupta, and J-P Talpin. A verification approach for gals integration of synchronous components. *Electronic Notes in Theoretical Computer Science*, 146(2) :105–131, 2006.

[Dru09] Jon Drummond. Understanding interactive systems. *Organised Sound*, 14(02) :124–133, 2009.

[DS90] Jacques Duthen and Marco Stroppa. Une représentation de structures temporelles par synchronisation de pivots. In *Proceedings of Colloque "Musique et Assistance Informatique*, pages 471–479, 1990.

[Dub06] Marc Duby. *Soundpainting as a system for the collaborative creation of music in performance*. PhD thesis, University of Pretoria, 2006.

[ECGJ11] José Echeveste, Arshia Cont, Jean-Louis Giavitto, and Florent Jacquemard. Formalisation des relations temporelles entre une partition et une performance musicale dans un contexte d'accompagnement automatique. In *Colloque Modélisation des Systèmes Réactifs (MSR)*, Lille, France, November 2011.

[EJCG13] José Echeveste, Florent Jacquemard, Arshia Cont, and Jean-Louis Giavitto. Operational Semantics of a Domain Specific Language for Real Time Musician-Computer Interaction. *Discrete Event Dynamic Systems*, 23 :343–383, December 2013.

[ELM+12] John C Eidson, Edward A Lee, Slobodan Matic, Sanjit A Seshia, and Jia Zou. Distributed real-time software for cyber–physical systems. *Proceedings of the IEEE*, 100(1) :45–59, 2012.

[Ess12] Georg Essl. *Playing with Time : Manipulation of Time and Rate in a Multi-rate Signal Processing Pipeline*. Ann Arbor, MI : MPublishing, University of Michigan Library, 2012.

[FJ13] Léa Fanchon and Florent Jacquemard. Formal Timing Analysis Of Mixed Music Scores. In *2013 ICMC - International Computer Music Conference*, Perth, Australia, August 2013.

[Fri95] Anders Friberg. *A Quantitative Rule System for Musical Performance*. PhD thesis, KTH - Royal Institute of Technology, Stockholm, Sweden, 1995.

[Gan95] Kyle Gann. *The Music of Conlon Nancarrow*. Cambridge University Press, 1995.

[GCE15] Jean-Louis Giavitto, Arshia Cont, and José Echeveste. Antescofo a not-so-short introduction to version 0.x. Technical report, Ircam, 2015.

[GD94] Lorin Grubb and Roger B Dannenberg. Automating ensemble performance. In *Proceedings of the International Computer Music Conference*, pages 63–63. INTERNATIONAL COMPUTER MUSIC ACCOCIATION, 1994.

[GE+14] Jean-Louis Giavitto, José Echeveste, et al. Real-time matching of antescofo temporal patterns. In *Proceedings*

*of the 16th International Symposium on Principles and Practice of Declarative Programming*, 2014.

[Gil74] Kahn Gilles. The semantics of a simple language for parallel programming. In *In Information Processing'74 : Proceedings of the IFIP Congress*, volume 74, pages 471–475, 1974.

[Gri81] Paul Griffiths. A guide to electronic music. *Music and Letters*, 62(3-4) :482–482, 1981.

[GS90] C. A. Gunter and D. S. Scott. *Handbook of Theoretical Computer Science*, volume B : Formal Models and Semantics, chapter Semantic Domains. MIT Press, Cambridge, MA, USA, 1990.

[GSVK+06] Arkadeb Ghosal, Alberto Sangiovanni-Vincentelli, Christoph M. Kirsch, Thomas A. Henzinger, and Daniel Iercan. A hierarchical coordination language for interacting real-time tasks. In *Proceedings of the 6th ACM &Amp ; IEEE International Conference on Embedded Software*, EMSOFT '06, pages 132–141, New York, NY, USA, 2006. ACM.

[Gué05] Yann Guédon. Hidden hybrid markov/semi-markov chains. *Computational statistics & Data analysis*, 49(3) :663–688, 2005.

[Hal98] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991.

[Hon01] Henkjan Honing. From time to time : The representation of timing and tempo. *Computer Music Journal*, 25(3) :50–61, 2001.

[Hud04] Paul Hudak. An algebraic theory of polymorphic temporal media. In Bharat Jayaraman, editor, *Practical Aspects of Declarative Languages*, volume 3057 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2004.

[Hud14] Paul Hudak. *The Haskell School of Music – From Signals to Symphonies*. (Version 2.6), January 2014.

[Jaf85] David Jaffe. Ensemble timing in computer music. *Computer Music Journal*, pages 38–48, 1985.

[JGAK07] Sergi Jordà, Günter Geiger, Marcos Alonso, and Martin Kaltenbrunner. The reactable : Exploring the synergy between live music performance and tabletop tangible interfaces. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, TEI '07, pages 139–146, New York, NY, USA, 2007. ACM.

[JLM14] Fatma Jebali, Frédéric Lang, and Radu Mateescu. Grl : A specification language for globally asynchronous locally synchronous systems. In *Formal Methods and Software Engineering*, pages 219–234. Springer, 2014.

[KM09] Alexis Kirke and Eduardo Reck Miranda. A survey of computer systems for expressive music performance. *ACM Comput. Surv.*, 42(1) :3 :1–3 :41, December 2009.

[Kop91] Hermann Kopetz. Event-Triggered versus Time-Triggered Real-Time Systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, volume 563, pages 87–101, Dagstuhl Castle, Germany, July 8-12 1991. Springer.

[KZ87] Tag Gon Kim and Bernard P Zeigler. The devs formalism : hierarchical, modular systems specification in an object oriented framework. In *Proceedings of the 19th conference on Winter simulation*, pages 559–566. ACM, 1987.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, July 1978.

[Lan64] Peter J Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4) :308–320, 1964.

[Lar01] Edward W. Large. Periodicity, Pattern Formation, and Metric Structure. *Journal of New Music Research*, 22 :173–185, 2001.

[Lau89] Mikael Laurson. *PATCHWORK : a Graphic Language in PREFORM*. Ann Arbor, MI : MPublishing, University of Michigan Library, 1989.

[LBA12] Benjamin Lévy, Georges Bloch, and Gérard Assayag. OMaxist dialectics. In *Proc. of the International Conference on New Interfaces for Musical Expression*, pages 137–140, 2012.

[LDFW87] Insup Lee, Susan B Davidson, and Victor Fay-Wolfe. Motivating time as a first class entity. 1987.

[Lee07] Eric Lee. A semantic time framework for interactive media systems. Master's thesis, RWTH Aachen University, Aachen, Germany, September 2007.

[Lee08] Edward A. Lee. Cyber physical systems : Design challenges. Technical Report UCB/EECS-2008-8, EECS Department, University of California, Berkeley, Jan 2008.

[Lee14] Edward A. Lee. Constructive models of discrete and continuous physical phenomena. Technical Report UCB/EECS-2014-15, EECS Department, University of California, Berkeley, Feb 2014.

[Lew99] George E Lewis. Interacting with latter-day musical automata. *Contemporary Music Review*, 18(3) :99–112, 1999.

[LFL12] Zimu Liu, Yuan Feng, and Baochun Li. Musicscore : Mobile music composition for practice and fun. In *Proceedings of the 20th ACM international conference on Multimedia*, MM '12, pages 109–118, New York, NY, USA, 2012. ACM.

[LGLBLM91] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9) :1321–1336, 1991.

[LJ99] E.W. Large and M.R. Jones. The Dynamics of Attending : How People Track Time-Varying Events. *Psychological review*, 106(1) :119, 1999.

[LJ11] Palmer C. Loehr JD, Large EW. Temporal Coordination and Adaptation to Rate Change in Music Performance. *Journal of Experimental Psychology : Human Perception and Performance*, August 2011.

[LM86] Daniel Le Métayer. A new computational model and its discipline of programming. 1986.

[LXD11] Dawen Liang, Guangyu Xia, and Roger B Dannenberg. A framework for coordination and synchronization of media. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 167–172, 2011.

[LZ05] Edward A. Lee and Haiyang Zheng. Operational semantics of hybrid systems. In Manfred Morari and Lothar Thiele, editors, *HSCC*, volume 3414 of *Lecture Notes in Computer Science*, pages 25–53. Springer, 2005.

[LZ07] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT '07 : Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 114–123, New York, NY, USA, 2007. ACM.

[Man90] Philippe Manoury. *La note et le son*. L'Hamartan, 1990.

[Man12] Philippe Manoury. Les procédés de composition utilisés dans "tensio". 2012.

[McC96] James McCartney. A New Real-time Synthesis Language. In *Proceedings of International Computer Music Conference (ICMC)*, Hong Kong, Chine, 1996.

[Mil91] David L. Mills. Internet time synchronization : The network time protocol. *IEEE Transactions on Communications*, 1991.

[MMM+69] Max V Mathews, Joan E Miller, F Richard Moore, John R Pierce, and Jean-Claude Risset. *The technology of computer music*. MIT press Cambridge, 1969.

[MMP91] Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, pages 447–484. Springer, 1991.

[Mor08] Pierre-Etienne Moreau. *Programmation et confiance*. PhD thesis, Institut National Polytechnique de Lorraine-INPL, 2008.

[Mos90] P. D. Mosses. *Handbook of Theoretical Computer Science*, volume B : Formal Models and Semantics, chapter Denotational semantics. MIT Press, Cambridge, MA, USA, 1990.

[MP92] Zohar Manna and Amir Pnueli. Verifying hybrid systems. In Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 4–35. Springer, 1992.

[MP08] Louis Mandel and Florence Plateau. Interactive Programming of Reactive Systems. In *Proceedings of Model-driven High-level Programming of Embedded Systems (SLA++P'08)*, Electronic Notes in Computer Science,

pages 44–59, Budapest, Hungary, April 2008. Elsevier Science Publishers.

[MS10] John MacCallum and Andrew Schmeder. *Timewarp : A Graphical Tool For The Control Of Polyphonic Smoothly Varying Tempos*. Ann Arbor, MI : MPublishing, University of Michigan Library, 2010.

[MS14] Eleftherios Matsikoudis and Christos Stergiou. First draft of the act programming language. Technical report, DTIC Document, 2014.

[MSZ] Pieter J. Mosterman, Gabor Simko, and Justyna Z. A hyperdense semantic domain for discontinuous behavior in physical system models.

[Mur14] Jim Murphy. *Expressive musical robots : building, evaluating, and interfacing with an ensemble of mechatronic instruments : a thesis submitted to the Victoria University of Wellington in fulfilment of the requirements for the degree of Doctor of Philosophy*. PhD thesis, The Author, 2014.

[MZ94] Guerino Mazzola and Oliver Zahorka. Tempo Curves Revisited : Hierarchies of Performance Fields. *Computer Music Journal*, 18(1) :40–52, 1994.

[NC15] Jérôme Nika and Marc Chemillier. Improvisation musicale homme-machine guidée par un scénario temporel. *Technique Et Science Informatique, Numéro Spécial RenPar\'13*, 7(33) :651–684, 2015.

[NECG14] Jérôme Nika, José Echeveste, Marc Chemillier, and Jean-Louis Giavitto. Planning Human-Computer Improvisation. In *Proceedings of ICMC-SMC*, 2014.

[Nou08] Gilbert Nouno. *Suivi de tempo appliqué aux musiques improvisées, à la recherche du temps perdu...* PhD thesis, Paris 6, 2008.

[OFL09] Yann Orlarey, Dominique Fober, and Stephane Letz. *FAUST : an Efficient Functional Approach to DSP Programming*, pages 65–96. 2009.

[Pac03] Francois Pachet. The continuator : Musical interaction with style. *Journal of New Music Research*, 32(3) :333–341, 2003.

[Pal97] Caroline Palmer. Music performance. *Annual Review of Psychology*, 48 :155–138, 1997.

[PJ15] Clément Poncelet and Florent Jacquemard. Model based testing of an interactive music system. In *ACM SAC*, 2015.

[Pou06] M. Pouzet. *Lucid Synchrone, version 3*. Université Paris-Sud, LRI, 2006.

[Pre07] William H Press. *Numerical recipes 3rd edition : The art of scientific computing*. Cambridge university press, 2007.

[PRMd13] François Pachet, Pierre Roy, Julian Moreira, and Mark d'Inverno. Reflexive loopers for solo musical improvisation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2205–2208. ACM, 2013.

[PSJ14] Clément Poncelet Sanchez and Florent Jacquemard. Test Methods for Score-Based Interactive Music Systems. In *Proceedings of the ICMC-SMC*, Athen, Grèce, 2014.

[PT08] Wolfgang Pree and Josef Templ. Modeling with the timing definition language (tdl). In Manfred Broy, IngolfH. Krüger, and Michael Meisinger, editors, *Model-Driven Development of Reliable Automotive Services*, volume 4922 of *Lecture Notes in Computer Science*, pages 133–144. Springer Berlin Heidelberg, 2008.

[Pto14] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[ptp05] *Software and hardware prototypes of the IEEE 1588 precision time protocol on wireless LAN Software and hardware prototypes of the IEEE 1588 precision time protocol on wireless LAN*, 2005.

[Puc91] Miller Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. In *Proceedings of International Computer Music Conference (ICMC)*, volume 15, pages 68–77, Montréal, Canada, 1991.

[Puc02] Miller Puckette. Using pd as a score language. In *Proc. Int. Computer Music Conf*, pages 184–187, 2002.

[Puc04] Miller Puckette. A divide between 'compositional'and 'performative'aspects of pd. In *Proc. First International Pd Convention*, 2004.

[PV01] Catuscia Palamidessi and Frank D Valencia. A temporal concurrent constraint programming calculus. In

*Principles and Practice of Constraint Programming—CP 2001*, pages 302–316. Springer, 2001.

[Rap11] Christopher Raphael. The Informatics Philharmonic. *Commun. ACM*, 54 :87–93, March 2011.

[Rep06] BrunoH. Repp. Musical synchronization. *Music, motor control, and the brain*, 129(10) :55–76, 2006.

[Ris99] Jean-Claude Risset. Composing in real-time ? *Contemporary Music Review*, 18(3) :31–39, 1999.

[Rob09] Andrew Robertson. *Interactive real-time musical systems*. PhD thesis, School of Electronic Engineering and Computer Science, Queen Mary University of London, 2009.

[Roe03] Axel Roebel. Transient detection and preservation in the phase vocoder. In *International Computer Music Conference (ICMC)*, pages 247–250, Singapore, Singapore, Octobre 2003.

[Row92] Robert Rowe. *Interactive music systems : machine listening and composing*. MIT Press, Cambridge, MA, USA, 1992.

[RR05] Axel Roebel and Xavier Rodet. Efficient spectral envelope estimation and its application to pitch shifting and envelope preservation. In *International Conference on Digital Audio Effects*, pages 30–35, Madrid, Spain, Septembre 2005.

[RS13] BrunoH. Repp and Yi-Huang Su. Sensorimotor synchronization : A review of recent research (2006–2012). *Psychonomic Bulletin & Review*, 20(3) :403–452, 2013.

[SBVB06] Diemo Schwarz, Gregory Beller, Bruno Verbrugghe, and Sam Britton. Real-time corpus-based concatenative synthesis with catart. In *9th Int. Conference on Digital Audio Effects (DAFx-06)*, Montreal, Canada, September 2006.

[SD13] Greg Surges and Shlomo Dubnov. Feature selection and composition using pyoracle. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[sem15] Mutant real-time multimedia computing seminars. http://repmus.ircam.fr/mutant/rtmseminars, mars 2015.

[SG10] Andrew Sorensen and Henry Gardner. Programming With Time : Cyber-Physical Programming With Impromptu. In *ACM Sigplan Notices*, volume 45, pages 822–834. ACM, 2010.

[SN04] Ralf Steinmetz and Klara Nahrstedt. *Multimedia systems*. Springer Science & Business Media, 2004.

[Sor05] Andrew Sorensen. Impromptu : An interactive programming environment for composition and performance. In *Proceedings of the Australasian Computer Music Conference 2009*, 2005.

[SR89] Vijay A Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM, 1989.

[Str99] Marco Stroppa. Live electronics or... live music ? towards a critique of interaction. *Contemporary Music Review*, 18(3) :41–77, 1999.

[Tau91] Heinrich Taube. Common music : A music composition language in common lisp and clos. *Computer Music Journal*, pages 21–32, 1991.

[TE14] Christopher Trapani and José Echeveste. Real time tempo canons with antescofo. In *International Computer Music Conference*, page 207, 2014.

[Thooo] Margaret E. Thomas. Nancarrow's canons : Projections of temporal and formal structures. *Perspectives of New Music*, 38(106-133), Summer 2000.

[Tif94] Vincent Tiffon. *Recherches sur les musiques mixtes*. PhD thesis, Aix-Marseille 1, 1994.

[Tod95] Neil P. McAngus Todd. The kinematics of musical expression. *The Journal of the Acoustical Society of America*, 97(3) :1940–1949, 1995.

[Tru07] Dan Trueman. Why a laptop orchestra ? *Organised Sound*, 12(02) :171–179, 2007.

[Ver84] Barry Vercoe. The Synthetic Performer in the Context of Live Performance. In *Proceedings of International Computer Music Conference (ICMC)*, pages 199–200, 1984.

[vid13] Real-time tempo canons with antescofo jérôme conte preformance. https://www.youtube.com/watch?v=CQ1pQNy5XJk, April 2013.

[Vit63] A.J. Viterbi. Phase-locked loop dynamics in the presence of noise by fokker-planck techniques. *Proceedings of the IEEE*, 51(12) :1737–1753, Dec 1963.

[VP85] Barry Vercoe and Miller Puckette. *Synthetic rehearsal : Training the synthetic performer*. Ann Arbor, MI : MPublishing, University of Michigan Library, 1985.

[Wan08] Ge Wang. *The ChucK Audio Programming Language : An Strongly-timed and On-the-fly Environ/mentality*. PhD thesis, Princeton University, 2008.

[WEBV14] Alan M. Wing, Satoshi Endo, Adrian Bradbury, and Dirk Vorberg. Optimal Feedback Correction in String Quartet Synchronization. *Journal of The Royal Society Interface*, 11(93), 2014.

[WG04] Gerhard Widmer and Werner Goebl. Computational models of expressive music performance : The state of the art. *Journal of New Music Research*, 33(3) :203–216, 2004.

[Win01] Todd Winkler. *Composing interactive music : techniques and ideas using Max*. MIT press, 2001.

[ZLL07] Yang Zhao, Jie Liu, and Edward A Lee. A programming model for time-synchronized distributed real-time systems. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS'07. 13th IEEE*, pages 259–268. IEEE, 2007.

[ZPP15] Anna Zamm, PeterQ. Pfordresher, and Caroline Palmer. Temporal coordination in joint music performance : effects of endogenous rhythms and auditory feedback. *Experimental Brain Research*, 233(2) :607–615, 2015.

Cinquième partie

# ANNEXES

# A

# TRADUCTION DE LA SÉMANTIQUE EN CAML

```
(* --- services --- *)

let fail_with x =
  Printf.printf "\n#########\nFail : %s\n" x;
  print_string x;
  failwith x

(*print functions*)

let rec prl pr prefix infix suffix l =
  Printf.printf "%s" prefix; prlaux pr infix suffix l

and prlaux pr infix suffix = function
  | [] -> Printf.printf "%s" suffix
  | [a] -> Printf.printf "%a%s" pr a suffix
  | a::l -> Printf.printf "%a%s" pr a infix; prlaux pr infix
      suffix l

and prl0 pr prefix infix suffix out l =
  let pr0 out = pr
  in prl pr0 prefix infix suffix l


(* generic lexicographic comparison *)
let cmpl_aux cmpx x1 x2 cmpl l1 l2 =
  if cmpx x1 x2 then true
  else if cmpx x2 x1 then false else cmpl l1 l2

let rec cmpl cmp l1 l2 = match l1, l2 with
  | [], [] -> false
  | [], _ -> true
  | _, [] -> false
  | a::l1', b::l2' -> cmpl_aux cmp a b (cmpl cmp) l1' l2'

(* auxiliary function for next : split a list in a pair minimal
    element and rest of the list *)
let split_min cmp =
  let rec mfct m rest = function
    | [] -> (m, rest)
    | a::l -> if cmp a m then mfct a (m::rest) l else mfct m (a::
        rest) l
  in function
    | [] -> fail_with "min_of: empty set"
    | a::l -> mfct a [] l
```

```
let freshuid =
  let cpt = ref 0 in function () -> (incr cpt; !cpt)

let freshid () = "id'" ^(string_of_int (freshuid ()))



(* --- Value, Expression and eval functions --- *)

type value =
  | Float of float
  | Bool of bool
  | String of string
  | Fct of (value -> value)

type environ = string -> value

let pr_v = function
  | Float f -> Printf.printf "%f" f
  | Bool true -> Printf.printf "true"
  | Bool false -> Printf.printf "false"
  | String s -> Printf.printf "%s" s
  | Fct _ -> Printf.printf "<fun>"

let pr_v0 out = pr_v


type expr =
  | Fail
  | Undef of string (* special form to test if a variable is
      undefined *)
  | Val of value
  | Var of string
  | Apply of expr * expr
  | Delay of expr


let is_a_cte_float_expression  = function
  | Val(Float _) -> true
  | Delay(Val(Float _)) -> true
  | _ -> false

and get_cte_float = function
  | Val(Float f) -> f
  | Delay(Val(Float f)) -> f
  | _ -> failwith "expression is not a cte float"

let rec pr_exp = function
  | Fail -> Printf.printf "fail"
```

```
  || Undef id -> Printf.printf "undef(%s)" id
  || Val v -> pr_v v
  || Var id -> Printf.printf "%s" id
  || Apply (f, a) -> Printf.printf "%a(%a)" pr_exp0 f pr_exp0 a
  || Delay d -> Printf.printf "[%a]" pr_exp0 d

and pr_exp0 out = pr_exp

let get_bool = function
  || Bool x -> x
  || _ -> fail_with "not a bool"

let get_float = function
  || Float x -> x
  || _ -> fail_with "not a float"

let apply1 f a = match f with
       || Fct fct -> fct a
       || _ -> fail_with "eval: Apply: not a function"


exception Var_not_found of string

let rec eval env = function
  || Fail -> fail_with "_||_"
  || Undef id ->
    let ret =
    (try ignore(env id); false with Var_not_found _ -> true)
    in Bool ret
  || Val v -> v
  || Var id -> env id
  || Apply (f, a) -> apply1 (eval env f) (eval env a)
  || Delay d ->
    (match (eval env d) with
    || (Float d') -> Float d'
    || _ -> fail_with "eval: Delay: bad duration")


(* auxiliary functions to build expressions *)

let apply2 f a b  = match f with
       || Fct fct -> apply1 (fct a) b
       || _ -> fail_with "eval: Apply2: not a function"

let _vplus = function
  || Float f1 -> Fct (function
      || Float f2 -> Float (f1 +. f2)
      || _ -> fail_with "aplus: bad argument type")
  || Bool b1 -> Fct (function
      || Bool b2 -> Bool (b1 ||| b2)
      || _ -> fail_with "plus: bad argument type")
  || _ -> fail_with "plus: bad argument type"
```

```
let vplus = Fct _vplus

let _vmoins = function
  | Float f1 -> Fct (function
      | Float f2 -> Float (f1 -. f2)
      | _ -> fail_with "moins: bad argument type")
  | _ -> fail_with "moins: bad argument type"

let vmoins = Fct _vmoins

let _vtimes = function
  | Float f1 -> Fct (function
      | Float f2 -> Float (f1 *. f2)
      | _ -> fail_with "times: bad argument type")
  | Bool b1 -> Fct (function
      | Bool b2 -> Bool (b1 && b2)
      | _ -> fail_with "times: bad argument type")
  | _ -> fail_with "times: bad argument type"

let vtimes = Fct _vtimes

let _vinf = function
  | Float f1 -> Fct (function
      | Float f2 -> Bool (f1 < f2)
      | _ -> Bool false)
  | Bool true -> Fct (function _ -> Bool false)
  | Bool false -> Fct (function x -> x)
  | _ -> Bool false

let vinf = Fct _vinf

let _vgeq = function
  | Float f1 -> Fct (function
      | Float f2 -> Bool (f1 >= f2)
      | _ -> Bool false)
  | Bool true ->  Fct (function _ -> Bool true)
  | Bool false -> Fct (function Bool x -> Bool (not x) | _ ->
     Bool false)
  | _ -> Bool false

let vgeq = Fct _vgeq




(* --- par. 7.1 : Domains construction --- *)


type ('event, 'time) tsequence =
```

```
  || End
  || Passage of 'time * ('event, 'time) tsequence
  || Event of 'event * ('event, 'time) tsequence

let rec pr_tsequence pr_e pr_t = function
  || End -> Printf.printf "end"
  || Passage (t, s) -> pr_t t; Printf.printf " ; "; pr_tsequence
     pr_e pr_t s
  || Event (e, s) -> pr_e e; Printf.printf " ; "; pr_tsequence
     pr_e pr_t s

type id = string

let rec bullet a b = match a with
  || End -> b
  || Passage (d, s) -> Passage (d, bullet s b)
  || Event (e, s) -> Event(e, bullet s b)




(* --- finite trace --- *)

(* par. 7.2 : only one constructor that corresponds to the
    assignment "id :=
 * value"
 *)
type event0 = Update0 of id * value

(* In the trace, time passage is given by a float (in secondes)
    and events correspond to assignmeent of variables *)

type trace = (event0, float) tsequence

let pr_event = function Update0 (id, v) -> Printf.printf "%s := "
     id; pr_v v
let pr_event0 out s = pr_event s

let pr_trace = pr_tsequence pr_event (function x -> Printf.printf
     "(%f s)" x)

let pr_trace0 out = pr_trace

(* par. 7.2 trace as environment *)

let rec lookTrace env x =
  Printf.printf "Look %s in %a\n" x pr_trace0 env;
  look env x

and look env x = match env with
  || End -> raise (Var_not_found x)
  || Passage(d, s) -> look s x
```

```
  || Event(Update0 (y, v), s) when x = y ⟶ look0 v x s
  || Event(_, s) ⟶ look s x

and look0 v x = function
  || End ⟶ v
  || Passage(d, s) ⟶  look0 v x s
  || Event(Update0 (y, nv), s) when x = y ⟶ look0 nv x s
  || Event(_, s) ⟶ look0 v x s

let rec undef env x = match env with
  || End ⟶ true
  || Passage(d, s) ⟶ undef s x
  || Event(Update0 (y, _), s) when x = y ⟶ false
  || Event(_, s) ⟶ undef s x


(* building traces *)
let add env x v = bullet env (Event(Update0(x, v), End))
let inc env x v = bullet env (Event(Update0(x, apply2 vplus (look
     env x) v), End))



let rec add_event x v = function
  || End ⟶ Printf.printf "————— %s := %a\n" x pr_v0 v ;
          Event(Update0(x, v), End)
  || Passage (d, s) ⟶ Passage (d, add_event x v s)
  || Event (e, s) ⟶ Event(e, add_event x v s)

let rec add_time i =  function
  || End ⟶ Printf.printf "— time passage %f\n" i;
          Passage(i, End)
  || Passage (d, s) ⟶ Passage (d, add_time i s)
  || Event (e, s) ⟶ Event(e, add_time i s)




(* --- par. 7.3 : temporal scopes --- *)

type uid_t = int
type labels_t = string list
type scope_t = Global || Local

type sync_t = || Loose
              || Tight
              || DTarget of float (* dumping factor *)
              || STarget of float (* position *)

type temporal_scope = {
  uid : uid_t option;
```

```
  tempo : expr option;
  sync : sync_t option;
  scope : scope_t option;
  beatPos : float option;
  expEvent : float option;
  labels : labels_t option;
}


let get_gen msg access ts = match access ts with
 ||Some x ⟶ x
 ||None ⟶ failwith ("undef in temporal scope: " ^ msg)

let is_gen access ts = match access ts with
 ||Some _ ⟶ true
 ||None ⟶ false

let get_uid = get_gen "uid" (function x ⟶ x.uid)
let is_uid = is_gen (function x ⟶ x.uid)

let get_tempo = get_gen "tempo" (function x ⟶ x.tempo)
let is_tempo = is_gen (function x ⟶ x.tempo)

let get_sync = get_gen "sync" (function x ⟶ x.sync)
let is_sync = is_gen (function x ⟶ x.sync)

let get_scope = get_gen "scope" (function x ⟶ x.scope)
let is_scope = is_gen (function x ⟶ x.scope)

let get_beatPos = get_gen "beatPos" (function x ⟶ x.beatPos)
let is_beatPos = is_gen (function x ⟶ x.beatPos)

let get_expEvent = get_gen "expEvent" (function x ⟶ x.expEvent)
let is_expEvent = is_gen (function x ⟶ x.expEvent)

let get_labels = get_gen "labels" (function x ⟶ x.labels)
let is_labels = is_gen (function x ⟶ x.labels)


let merge_ts t1 t2 =
{
        uid = if (is_uid t2) then t2.uid else t1.uid ;
        tempo = if (is_tempo t2) then t2.tempo else t1.tempo ;
        sync = if (is_sync t2) then t2.sync else t1.sync ;
        scope = if (is_scope t2) then t2.scope else t1.scope ;
        beatPos = if (is_beatPos t2) then t2.beatPos else t1.
            beatPos ;
        expEvent = if (is_expEvent t2) then t2.expEvent else t1.
            expEvent ;
        labels = if (is_labels t2) then t2.labels else t1.labels
            ;
}
```

```
let avance_time d ts =
  {
    uid = ts.uid;
    tempo = ts.tempo;
    sync = ts.sync;
    scope = ts.scope;
    beatPos = Some ((get_beatPos ts) +. d);
    expEvent = ts.expEvent;
    labels = ts.labels
  }

let update_expEvt ts pos =(* -1. means undefined *)
  {
    uid = ts.uid;
    tempo = ts.tempo;
    sync = ts.sync;
    scope = ts.scope;
    beatPos = ts.beatPos;
    expEvent = if pos < 0. then None else Some pos;
    labels = ts.labels
  }


let fresh_uid =
  let cpt = ref 2
  in function ts ⟶
  {
        uid = (incr cpt; Some !cpt);
        tempo = ts.tempo ;
        sync = ts.sync ;
        scope = ts.scope ;
        beatPos = ts.beatPos ;
        expEvent = ts.expEvent ;
        labels = ts.labels ;
  }

let tau0 =
 {
        uid = Some (-1);
        tempo = Some (Val (Float 1.));
        sync = Some Loose;
        scope = Some Global;
        beatPos = Some 0.;
        expEvent = None;
        labels = Some [];
}



let not_exp_ev ts = not (is_expEvent ts)
```

```
let string_sync = function
  || Loose ⟶ "loose"
  || Tight ⟶ "tight"
  || DTarget (f) ⟶ "dyn_target " ^(string_of_float f)
  || STarget (f) ⟶ "static_target " ^(string_of_float f)

(* 𝒟 *)
type delay = expr

(* 𝒜 *)

type statement =
  || Abort of string
  || Update of string * expr
  || Reaction of expr * string list * program
  || Group of program * temporal_scope

(* 𝒫 *)
and program = (statement, delay) tsequence


type result = Time of delay || Action of statement

let hd = function
  || End ⟶ fail_with "hd: empty program"
  || Passage (d, _) ⟶ Time d
  || Event(e, _) ⟶ Action e

and tl = function
  || End ⟶  fail_with "tl: empty program"
  || Passage (_, s) ⟶ s
  || Event(_, s) ⟶ s


let eval_delay d' env = function
  || Float f ⟶ d' /. f
  || x ⟶
    Printf.printf "### Delta: bad argument:\n\t%a\n" pr_v0 x;
    failwith "Not Yet implemented : arbirary tempo expr"

let delta ts env d =
  match eval env d with
  || Float f ⟶
    if (f < 0. ) then failwith "negative delay"
    else eval_delay f env (eval env (get_tempo ts))
  || x ⟶
    failwith "non numeric delay"



(* par. 7.3 :  <ρ total ordering of programs *)
```

```
let rec cmp_prog env (p1, r1) (p2, r2) = match p1, p2 with
  || End, End ⟶ false
  || End, _ ⟶ true
  || _, End ⟶ false

  || _ ⟶  if(is_expEvent r1)
              && (get_expEvent r1) < (get_float (env "pos") )
                && ( (not (is_expEvent r2)) ||| (get_expEvent r1)
                    < (get_expEvent r2))
          then true
    else cmp_prog2 env (p1, r1) (p2, r2)

and cmp_prog2 env (p1, r1) (p2, r2) = match p1, p2 with
  || End, _ ⟶ failwith "internal error"
  || _, End ⟶ failwith "internal error"
  || Event(e1, s1), Event(e2, s2) ⟶ r1.uid < r2.uid

  || Passage(d1, s1), Passage (d2, s2) ⟶
    let f1 = delta r1 env d1
    and f2 = delta r2 env d2
    in if (f1 < f2) then true
       else if (f1 > f2) then false
       else r1.uid < r2.uid

  || Passage(d, s1), Event(e, s2)
   ⟶ let f =  delta r1 env d
       in if (f = 0.) then cmp_prog env (s1, r1) (p2, r2)
          else false

  || Event(e, s1), Passage(d, s2)
   ⟶ let f =  delta r2 env d
       in if (f = 0.) then cmp_prog env (p1, r1) (s2, r2)
          else true


(* printing of a program *)
let pr_delay d = Printf.printf "%a" pr_exp0 d

let pr_delay0 out = pr_delay

let rec pr_action : statement ⟶ unit = function
  || Abort s ⟶ Printf.printf "abort %s" s
  || Update (id, exp) ⟶ Printf.printf "%s := %a" id pr_exp0 exp
  || Group (p, ts) ⟶ Printf.printf "%a:(%a)" pr_ts0 ts pr_prog0 p
  || Reaction (cond, idl, p) ⟶
    Printf.printf "when:%a: %a ⟶%a"
                  (prl0 print_string "" ", " "") idl
                  pr_exp0 cond
                  pr_prog0 p

and pr_action0 _ x = pr_action x
```

```
and pr_ts  ts =
  Printf.printf "[uid=%s, beatPos=%s, expEvt=%s, tempo=%s, scope
     =%s, sync=%s, labels=%a]"
    (if is_uid ts then string_of_int (get_uid ts) else "undef")
    (if is_beatPos ts then string_of_float (get_beatPos ts) else
        "undef")
    (if is_expEvent ts then string_of_float (get_expEvent ts)
       else "undef")
    (match ts.tempo with
    | Some(Val(Float t)) -> string_of_float t
    | _ -> "undef")
    (if is_scope ts then (if (get_scope ts) = Global then "global
        " else "local") else "undef")
    (string_sync (get_sync ts))
    (prl0 print_string " " " " "") (get_labels ts)

and pr_ts0 out ts = pr_ts ts


and pr_prog p = pr_tsequence pr_action pr_delay p

and pr_prog0 out x = pr_prog x

and pr_prog1 out (x, t) = Printf.printf "%a : (%a)" pr_ts0 t
    pr_prog0 x

and pr_statement0 out (x, t) = Printf.printf "%a : (%a)" pr_ts0 t
     pr_action0 x

and pr_statementset0 out l = prl pr_statement0 "{ " " ||| " " }" l

and pr_progset l = prl pr_prog0 "{ " " ||| " " }" l

and pr_progset0 out x = pr_progset x

and pr_progset1 out x = prl pr_prog1 "{ " " ||| " " }" x

let rec pri = function
  | End -> []
  | Passage (_, s) -> pri s
  | Event(Group (p, ts), s) ->
        (if(is_uid ts) then [get_uid ts] else []) @ (pri p) @ (
           pri s)
  | Event(Reaction(_, _, p), s) -> (pri p) @ (pri s)
  | Event(e, s) -> pri s

let pri2 (p, _) = pri p


let well_formed_prog p =
  let pl = pri p
  in let pls = List.sort compare pl
```

```
      in (List.length pl) = (List.length pls)

let well_formed_set ps =
  let pls = List.map pri2 ps
  in let plsf = List.flatten pls
     in let plsfs = List.sort compare plsf
        in (List.length plsf) = (List.length plsfs)

let well_formed_statement_set rs =
  let rls = List.map (function (Reaction(_, _, q), tau) ⟶
      get_uid(tau) :: (pri q)) rs
  in let rlsf = List.flatten rls
     in let rlsfs = List.sort compare rlsf
        in (List.length rlsf) = (List.length rlsfs)


(* input trace to a program *)

let rec t2p0 = function
  || End ⟶ End
  || Passage (d, s) ⟶ Passage(Delay(Val (Float d)), t2p0 s)
  || Event (Update0(id, v), s) ⟶ Event(Update(id, Val v), t2p0 s)

  and t2p input =
      (t2p0 input,
       { uid = Some (0);
         tempo = Some (Val(Float 1.));
         sync = Some Loose;
         scope = Some Global;
         beatPos = Some 0.;
         expEvent = None; (* negatif si undef *)
         labels = Some[];
       })

(* par. 7.5.2 *)
let rec next env ps =
  assert (well_formed_set ps);
  match ps with
  || [] ⟶ fail_with "next: empty program set"
  || ps ⟶ let ((p, tau), rest) = split_min (cmp_prog (look env))
      ps
          in if p = End then next env rest
             else (hd p, tl p, tau, rest)

(* par. 7.5.5 *)

let inBeats env ts d = match eval env (get_tempo ts) with
  || Float f ⟶ f *. (delta ts env d)
  || Fct _ ⟶ fail_with "inBeats: general tempo fct not yet
     implemented"
  || _ ⟶ fail_with "inBeads: bad tempo specification"
```

```
let inBeats0 env d t = d *. (get_float (eval env t))



(* score is represented as a list of positions accessed through a
     global variable *)

let score = ref ([] : float list)

(* par. 7.5.3 *)

let rec precScore q = precScore0 q !score

and precScore0 q = function
  || [] ⟶ fail_with "precScore: no position less than the
       argument"
  || m::l ⟶ if m <= q then precScore1 q m l else precScore0 q l

and precScore1 q m = function
  || [] ⟶ m
  || r::l ⟶ if (r <= q) && (r > m) then  precScore1 q r l else
       precScore1 q m l

let prSc env d t = precScore ((get_float (look env "now")) +. (
    inBeats (look env) d t))

let rec sucScore0 pos = function
  || [] ⟶ 66666666666. (* end of times *)
  || m::l ⟶ if m > pos then m else sucScore0 pos l

let sucScore pos = sucScore0 pos !score


(* par. 7.5.7 *)

let labeled id (p, tau)  =
        if (not (is_labels tau))
                then false
                else List.mem id (get_labels tau)

let erase id ps =
List.filter (fun x ⟶ not (labeled id x)) ps


(* managing temporal scopes *)

let forget_expevt ts =
{
    uid = ts.uid;
    tempo = ts.tempo;
    sync = ts.sync;
    scope = ts.scope;
```

```
    beatPos = ts.beatPos;
    expEvent = if(ts.sync = Some Loose) then None else ts.
        expEvent;
    labels = ts.labels
  }

let set_expevt ts =
{
    uid = ts.uid;
    tempo = ts.tempo;
    sync = ts.sync;
    scope = ts.scope;
    beatPos = ts.beatPos;
    expEvent = if(ts.sync = Some Tight) then ts.beatPos else None
        ;
    labels = ts.labels
}

(* auxiliary functions for time pssage (C2) *)

let passage0 env dsec = function
  | (End, _) -> fail_with "passage: terminated programs in P"
  | (Event _, _) -> fail_with "passage: program starting with an
      event in P"
  | (Passage(Delay(d), s), ts) ->
      let d' = (inBeats env ts dsec) in
      let dres = apply2 vmoins (eval env d) (Float d')
      and ts' = avance_time d' ts in
        let ts'' = forget_expevt ts'
      in (Passage (Delay (Val dres), s), ts'')


let passage env dsec ps = List.map (passage0 env dsec) ps


(* auxiliary functions for update action (C3) *)

let ok_reaction rbar id env = function
  | (Reaction(ex, idl, _), _) as r
    -> (List.mem id idl) && not (List.mem r rbar) && get_bool(
        eval env ex)
  | (x, _)
    -> Printf.printf "get_reactions: bad set: %a\n" pr_action0 x;
    failwith "Bad formed R"

let get_reactions r rbar id env = List.filter (ok_reaction rbar
    id env) r


(* par. 7.5.6 *)

let evalFirstDelay env s = match hd s with
```

```
        || Time d -> Passage(Delay (Val (eval (look env) d)), (tl
            s))
        || _ -> s

(* --- semantics equations --- *)


(* initial environment *)

let env0 =
  Event(Update0("now", Float 0.),
        Event(Update0("rnow", Float 0.),
              Event(Update0("pos", Float 0.),
                    Event(Update0("tempo", Float 1.),  (* 0.5 = 2
                         fois + lentement *)
                          End))))

(* auxiliary function to create a unique variable *)
let make_once =
  let cpt = ref 1
  in (function () -> incr cpt; ("once_" ^ (string_of_int !cpt)))

(* C2 equation *)

let rec time_passage p r env dmin s ts ps' =
  Printf.printf "\t--- time passage %a\n" pr_exp0 dmin;
  assert (is_a_cte_float_expression dmin);
  let dsec = delta ts (look env) dmin
  in
     if dsec < 0. then failwith "negative delay"
     else
     let dmin_float = get_float (eval (look env) dmin) in
        let t = min (dmin_float *. (get_float(look env "tempo"))
            +.  get_float(look env "rnow")) (sucScore (get_float(
            look env "rnow"))) in
        let nenv0 = (add_time dmin_float env) in
          let nenv1 = add_event "rnow" (Float t) nenv0 in
            let nenv = add_event "now" (Float(dmin_float +.
                get_float(look env "now"))) nenv1
          and pp =  passage (look env) (Val(Float dsec)) ps'
          in if (s = End) then evalS pp r [] nenv
            else evalS ((s, forget_expevt (avance_time (
                get_cte_float dmin) ts)) :: pp) r [] nenv

(* par. 7.6 *)

and program_trace sc s input =
  score := sc;
  let ip = t2p input in
     assert (well_formed_prog (fst ip));
     assert (well_formed_statement_set s);
     let env = evalS [ip] s [] env0
```

```
     in Printf.printf "\ntrace resultante :\n\t%a\n" pr_trace0
         env;
        env

(* EvalE function in C3 *)

and evalE e (ps':(program * temporal_scope) list) r rbar env tau
    =
  match e with
  || Group (q, tau1) ⟶
       let p1 = evalFirstDelay env q and tau2 = set_expevt (
           fresh_uid (merge_ts tau tau1))
        in ((p1, tau2)::ps', r, rbar, env)

  || Abort id ⟶ (erase id ps', erase id r, erase id rbar, env)

  || Reaction _ ⟶ (ps', (e,tau)::r, rbar, env)

  || Update(id, exp) ⟶

    let v = eval (look env) exp in
    let envp = add_event id v env in
    let qr = get_reactions r rbar id (look envp) in
    let f = function
      || (Reaction (_, _, q), tau3)
              ⟶ (evalFirstDelay envp q, fresh_uid (merge_ts
                  tau tau3))
      || _ ⟶ failwith "evalE: badly formed qs"
    in let qp = List.map f qr
       in
        (qp @ ps', r, qr @ rbar, envp);


(* evalS function corresponds to the Eval function *)
and evalS p r rbar env =

  Printf.printf "      —— EvalS ——\n \t    P = %a\n\t   R = %a\n"
      (* "\tRbar = %a\n\t env = %a\n" *)
    pr_progset1 p
    pr_statementset0 r
    (*pr_statementset0 rbar
    pr_trace0 env *);

  assert (well_formed_set p);
  assert (well_formed_statement_set r);

  (****** C1 ******)

  if p = [] then env
  else
      let (e, s, tau, ps') = next env p
      in match e with
```

```
      (****** C2 ******)
      (* time passage *)

      | Time dmin when (not_exp_ev tau) ||| (get_expEvent tau) >=
          (get_float(look env "pos"))
               -> time_passage p r env dmin s tau ps'

      (****** C3 ******)
      (* statement *)

      | Action e
          when (None = tau.expEvent) ||| (get_float (look env "pos
              ")) <= (get_expEvent tau)
        -> Printf.printf "\t-- statement\n";
          let (pe, re, rbare, enve) = evalE e ps' r rbar env tau
              in
          let (pp, rp) = continuation e s ps' r enve tau in
             (match pe, pp with
              | [], []
              | [], [(End, _)]
              | [(End, _)], []
              | [(End, _)], [(End, _)]
                -> evalS [] (re @ rp) rbare enve
              | _, []
              | _, [(End, _)]
                -> evalS pe (re @ rp) rbare enve
              |[], _
              |[(End, _)], _
                -> evalS pp (re @ rp) rbare enve
              | _
                -> evalS (pe @ pp) (re @ rp) rbare enve
             )
      (****** C4 ******)
      (* Lateness Management *)
      | _ ->
        Printf.printf "\t-- lateness Management\n" ;
        evalLate e s ps' r rbar env tau


(* Compute P_p and R_p in par.  "Handling the Continuation."
 *)
and continuation e p ps' r enve tau =
  Printf.printf "CONTINUATION e=%a p=%a\n" pr_action0 e pr_prog0
      p;
  if p = End
  then ([], [])
  else match e with
  | Abort id when List.mem id (get_labels tau) -> ([], [])
  | _ ->
    match hd p with
```

```
    || Action _ (* elsewhere case :the rest do not begins with a
        delay *)
      ⟶ ([(p, tau)], [])
    || Time d ⟶
      let p' = tl p
      and d' = eval (look enve) d in
      let pos = (get_beatPos tau) +. (get_float d') in
      let newdel =  pos -. (get_float (look enve "rnow")) in
      let precEvent = precScore pos in
      let evtBefore = (precEvent > (get_float (look enve "rnow"))
          ) && ((get_beatPos tau) < precEvent)
      and once_id = make_once() in
      let expr_isUndef = Undef once_id
      and expr_proc_sup_precEvent =  Apply(Apply(Val vgeq, Var "
          pos"), Val (Float precEvent)) in
      let afterTime = Apply(Apply(Val vtimes,
          expr_proc_sup_precEvent), expr_isUndef)
      and drem = pos -. precEvent
      and taup = update_expEvt tau precEvent in
      let taur = update_expEvt (avance_time ((get_float d') -.
          drem) tau) precEvent
      in

         if (get_sync tau) = Loose
         then ([(Passage(Delay (Val d'), p'), tau)], [])
         else
           (assert(Tight = get_sync tau);
            if evtBefore then
              let update_once = Update(once_id, Val(Float 1.)) in
              let body = Event(update_once, Passage(Delay(Val(
                  Float drem)), p')) in
              let reaction = Reaction(afterTime, ["pos"], body)
              in  ([], [(reaction, taur)])
            else
              ([(Passage(Delay(Val (Float newdel)), p'), taup)],
                  [])
           )


and evalLate x p ps' r rbar env tau =
  match x with
  || Action e  when tau.scope = Some Global
    ⟶ let (pe, re, rbare, enve) = evalE e ps' r rbar env tau
      in
         if (p = End)
         then  evalS pe re rbare enve
         else  evalS ((p, tau) :: pe) re rbare enve

  || Action _ when tau.scope = Some Local
    ⟶ if (p = End) then  evalS ps' r rbar env
      else  evalS ((p, tau) :: ps') r rbar env
```

```
  || Time d ⟶
    let pose = look env "pos"
    and d' = eval (look env) d in
    let retard = (get_float pose) -. (get_expEvent tau) in
    let stayLate = (get_float d') < retard
    and pos = (get_expEvent tau) +. (get_float d') in
    let precEvent = precScore pos in
    let evtBefore = (precEvent > (get_float (look env "now")))
        && ((get_beatPos tau) < precEvent)
    and drem = pos -. precEvent
    and once_id = make_once() in
    let expr_isUndef = Undef once_id
    and expr_proc_sup_precEvent =  Apply(Apply(Val vgeq, Var "pos
        "), Val (Float precEvent)) in
    let afterTime = Apply(Apply(Val vtimes,
        expr_proc_sup_precEvent), expr_isUndef) in
    let taup1 = update_expEvt tau ((get_expEvent tau) +. (
        get_float d'))
    and taup2 = update_expEvt tau precEvent
    and taup3 = update_expEvt tau (-1.)
    and taur =  update_expEvt tau precEvent
    in
       if (not stayLate) && (tau.sync = Some Tight) && evtBefore
           then
         let update_once = Update(once_id, Val(Float 1.)) in
         let body = Event(update_once, Passage(Delay(Val(Float
             drem)), p)) in
         let reaction = Reaction(afterTime, ["pos"], body) in
         let r'p = (reaction, avance_time ((get_float d') -. drem
             ) taur)
         in  evalS ps' (r'p :: r) rbar env
       else (* in the remaining cases r'p is empty *)
         if stayLate
         then evalS ((p, avance_time (get_float d') taup1) :: ps
             ') r rbar env
         else
           if tau.sync = Some Tight
           then evalS ((Passage(Delay(Val d'), p), avance_time
               retard taup2) :: ps') r rbar env
           else evalS ((Passage(Delay(Val d'), p), avance_time
               retard taup3) :: ps') r rbar env


(* === EXAMPLES === *)


let input1 =
  Event(Update0("now", Float 0.),
        Event(Update0("pos", Float 0.),
              End))
let input2 =
```

```
  Event(Update0("now", Float 0.),
        Event(Update0("pos", Float 0.),
              Passage(2.3,
                      Event(Update0("pos", Float 2.),
                            End))))
(*let input3 =
  Event(Update0("now", Float 0.),
        Event(Update0("pos", Float 0.),
              Passage(1.3,
                      Event(Update0("pos", Float 2.),
                            End))))*)

  (*let input3 =
  Event(Update0("now", Float 0.),
        Event(Update0("rnow", Float 0.),
               Event(Update0("pos", Float 0.),
              Passage(1.3,
                      Event(Update0("rnow", Float 2.),
                        Event(Update0("pos", Float 2.),End))))))

*)
  let input3 =
  Event(Update0("now", Float 0.),
        Event(Update0("rnow", Float 0.),
               Event(Update0("pos", Float 0.),
                  Passage(1.2,
                    Event(Update0("rnow", Float 1.),
                      Event(Update0("pos", Float 1.),
                         Passage(1.3,
                          Event(Update0("rnow", Float 2.),
                            Event(Update0("pos", Float 2.),End)))
                              ))))))

let make_top n p =
  let cond1 =  Apply(Apply(Val vgeq, Var "pos"), Val (Float n))
  and oncevar = make_once () in
  let cond = Apply(Apply(Val vtimes, Undef oncevar), cond1)
  and once = (Update(oncevar, Val(Bool true)))
  and rttempo = Update("rt_tempo", Val(Float 1. ))
  and taun = {
                uid = Some (-1);
                tempo = Some (Var "rt_tempo");
                sync = Some Loose;
                scope = Some Global;
                beatPos = None;
                expEvent = Some n;
                labels = Some [];
            }
  in (Reaction(cond, ["pos"], Event(once,
                                    Event(rttempo, p))),
                taun)
```

```
let taug =
 {
        uid = Some (-1);
        tempo = Some (Val (Float 1.));
        sync = Some Tight;
        scope = Some Global;
        beatPos = None;
        expEvent = None;
        labels = Some ["group"];
}

let group1 = Event(Group(Event(Update("x", Var "rnow"),
                               Passage(Delay (Val(Float 1.5)),
                               (Event(Update("y", Var "rnow"),
                                   End)))),
                         taug),
                   End)
(*let group1 = Event(Group(Event(Update("x", Var "rnow"),
                               Event(Update("y", Var "rnow"),
                                   End)),
                         taug),
                   End)*)



(*let prog1 =
  let p = Passage(Delay (Val(Float 1.5)),
                  Event((Update("x", Var "now")),
                        End))
  in (make_top 0.0 p, tau0)
*)

let prog0 = (make_top 0.0 group1)

(*let prog0 =
  let p = Passage(Delay (Val(Float 1.7)),
                  group1)
  in (make_top 0.0 p)*)

let prog1 = (make_top 1.0 End)
let prog2 = (make_top 2.0 End)

let _ =
  Printf.printf "input = : %a\n" pr_prog0 (t2p0 input3)

let res = program_trace [0.0; 1.0; 2.0] [prog0;prog1;prog2]
    input3
;;
```